

# Database Processing

## CS 451 / 551

### Lecture 11: Transactions and Concurrency Control



**Suyash Gupta**

Assistant Professor

Distopia Labs and ONRG

Dept. of Computer Science

(E) [suyash@uoregon.edu](mailto:suyash@uoregon.edu)

(W) [gupta-suyash.github.io](https://github.com/suyashgupta)



**Assignment 2 is Out!**  
**Deadline: Nov 13, 2025 at 11:59pm**

**Assignment 3 will be released on Nov 13, 2025!**

# Transactions

- Transactions are ubiquitous!
- Examples: Banking, Online shopping, Trading, Social media, and so on.

# How to define a Transaction?

# How to define a Transaction?

- Transaction is a collection of operations.
- For example:
  - Moving money from one checkings account to savings account.
  - Buying a product from Amazon.

# How to define a Transaction?

- Transaction is a unit of program that reads and/or writes one or more data items.
- A common way to write a transaction in popular DBMS is by placing the body of the transaction between, “**begin transaction**” and “**end transaction**”.

# How to define a Transaction?

- Transaction is a unit of program that reads and/or writes one or more data items.
- A common way to write a transaction in popular DBMS is by placing the body of the transaction between, “**begin transaction**” and “**end transaction**”.
- Also, the reason why transaction is termed as an indivisible unit.
  - It either executes in its entirety or nothing at all.

# Definitions and Notations

- **Database:**
  - A collection of data-items or records (**A**, **B**, **C**, **D**, ...).
- **Transactions:**
  - A set of read/write operations:
  - **R(A)**  $\rightarrow$  implies Read a data-item/record **A**.
  - **W(A)**  $\rightarrow$  implies Write a data-item/record **A**.



# ACID Properties for a Transaction

?

# ACID Properties for a Transaction

- Each database should provide the following four properties for transactions :
- **Atomicity:** Either all operations of the transaction are reflected properly in the database, or none are.

# ACID Properties for a Transaction

- Each database should provide the following four properties for transactions:
- **Atomicity:** Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency:** Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.

# ACID Properties for a Transaction

- Each database should provide the following four properties for transactions:
- **Atomicity:** Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency:** Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.
  - Not referring to database consistency constraints.
- **Isolation:** For every pair of concurrent (executing at the same time) transactions  $T_i$  and  $T_j$ , either  $T_i$  finished execution before  $T_j$  started, or  $T_i$  started execution after  $T_j$  finished.
  - Transactions are unaware of other transactions executing

# ACID Properties for a Transaction

- **Atomicity:** Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency:** Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.
- **Isolation:** For every pair of concurrent (executing at the same time) transactions  $T_i$  and  $T_j$ , either  $T_i$  finished execution before  $T_j$  started, or  $T_i$  started execution after  $T_j$  finished.
  - Transactions are unaware of other transactions executing
- **Durability:** Once a transaction completes successfully, any changes it made to the database should persist, even if there are system failures.

# ACID Properties for a Transaction

- When do ACID properties come into play?

# ACID Properties for a Transaction

- When do ACID properties come into play?
  - Concurrency!
- In a concurrent system or database, two or more transactions may attempt to fetch the same data.
- But, why is this an issue?

# ACID Properties for a Transaction

- When do ACID properties come into play?
  - Concurrency!
- In a concurrent system or database, two or more transactions may attempt to fetch the same data.
- But, why is this an issue?
  - Concurrency if not handled well can lead to ACID violations.
  - For instance. → Race conditions!



# Two Concurrent Transactions

**T1:**

```
read(A);  
A = A - 50;  
write(A);  
read(B);  
B = B + 50;  
write(B).
```

**T2:**

```
read(A);  
temp = A * 0.1;  
A = A - temp;  
write(A);  
read(B);  
B = B + temp;  
write(B)
```

**Notice that they are accessing the same variables A and B.**

# Atomicity

- **What do you think happens at DBMS after you execute a transaction?**

# Atomicity

- What do you think happens at DBMS after you execute a transaction?
  - The transaction **commits** after completing all the operations!
  - The transaction **aborts**.

# Atomicity

- What do you think happens at DBMS after you execute a transaction?
  - The transaction **commits** after completing all the operations!
  - The transaction **aborts**.
- What is meant by transaction commits or aborts?

# Atomicity

- What do you think happens at DBMS after you execute a transaction?
  - The transaction **commits** after completing all the operations!
  - The transaction **aborts**.
- What is meant by a transaction commits or aborts?
  - **Commits** → The result of executing the transaction will persist.
  - **Aborts** → No trace of the transaction will exist.

# Atomicity

- **DBMS guarantees atomicity.**
  - From a user's point of view: a transaction always either executes all its operations or executes none at all.

# Atomicity

- How to provide support for atomicity in the DBMS?

# Atomicity

- How to provide support for atomicity in the DBMS?
- **Two ways:**
  - Logging
  - Shadow Paging (not preferred)



# Atomicity Support: Logging

# Atomicity Support: Logging

- A log is like a ledger or file that records all events or actions.
- DBMS logs every action so that it can undo the actions of aborted transactions.
- Essentially, you are noting down:
  - All the operations that you executed.
  - Any record in memory/disk you added/deleted/modified.
  - And the order of performing these operations.
- All of this information is also termed as **undo records**.
- You may have heard of the black box in airplanes → A form of logging.
- Audit Trail? → A form of logging.

# Atomicity Support: Shadow Paging

# Atomicity Support: Shadow Paging

- DBMS creates copies of each page.
- Effects of a transaction are applied to a specific copy.
- Only when the transaction **commits**, the page copy is made visible to others.
- Clearly, extremely bad in performance!
- Hard to maintain and page merge may be necessary.

# Consistency

- A transaction must preserve database consistency
  - If a transaction is run atomically in isolation starting from a consistent database, the database must again be consistent at the end of the transaction.
- Need to preserve the data integrity constraints like referential integrity, foreign key constraints, etc.
- Need to preserve application-dependent consistency constraints that are too complex to state using the SQL constructs for data integrity.
- **Responsibility of the programmer** who codes a transaction.
- Notice that the **C in ACID** is the only one that is not under the control of the system!

# Isolation

- Users submit transactions.
- Each transaction should execute as if it is running by itself.

# Isolation

- Users submit transactions.
- Each transaction should execute as if it is running by itself.
- But running one transactions at a time will give poor performance.

# Isolation

- Users submit transactions.
- Each transaction should execute as if it were running by itself.
- But **running one transactions at a time will give poor performance.**
- With the prevalence of multi-core architecture, DBMS should take advantage of the multiple cores.
- **Concurrency permits interleaving the transaction** operations.
  - Interleaving transactions also permits running one transaction when another is waiting for some resource (I/O, user input, or fetching data from disk).
  - Need a mechanism to interleave transactions but make it appear as if they ran one-at-a-time.



# Concurrent Transactions

- Assume that these two transactions are undergoing concurrent execution.
- What are the possible interleaving of these two transactions?

**T1:**

```
read(A);  
A = A - 50;  
write(A);  
read(B);  
B = B + 50;  
write(B).
```

**T2:**

```
read(A);  
A = A * 0.95;  
write(A);  
read(B);  
B = B * 1.05 ;  
write(B)
```

# Concurrent Transactions

- Before we determine possible inter-leavings, we need to do a bunch of tasks.
- First, we need to know the **possible set of values for A and B** at the end of running these transactions (Say, initially  $A = B = 50$ ):
  - $A + B = 100 * 1.05 = 105$

**T1:**

```
read(A);  
A = A - 50;  
write(A);  
read(B);  
B = B + 50;  
write(B).
```

**T2:**

```
read(A);  
A = A * 1.05;  
write(A);  
read(B);  
B = B * 1.05 ;  
write(B)
```

# Concurrent Transactions

- Next, we **transform these transactions** to the database perspective.
- Specifically, we need to worry only about read/write operations as only those impact the database.

**T1:**

```
read(A);  
A = A - 50;  
write(A);  
read(B);  
B = B + 50;  
write(B).
```

**T2:**

```
read(A);  
A = A * 0.95;  
write(A);  
read(B);  
B = B * 1.05 ;  
write(B)
```

# Concurrent Transactions

- We need to worry only about read/write operations as only those impact the database.
- So, we **re-write these transactions** as just a set of read/write operations.

**T1:**

**Begin**

read(A)

write(A)

read(B)

write(B)

**End**

**T2:**

**Begin**

read(A)

write(A)

read(B)

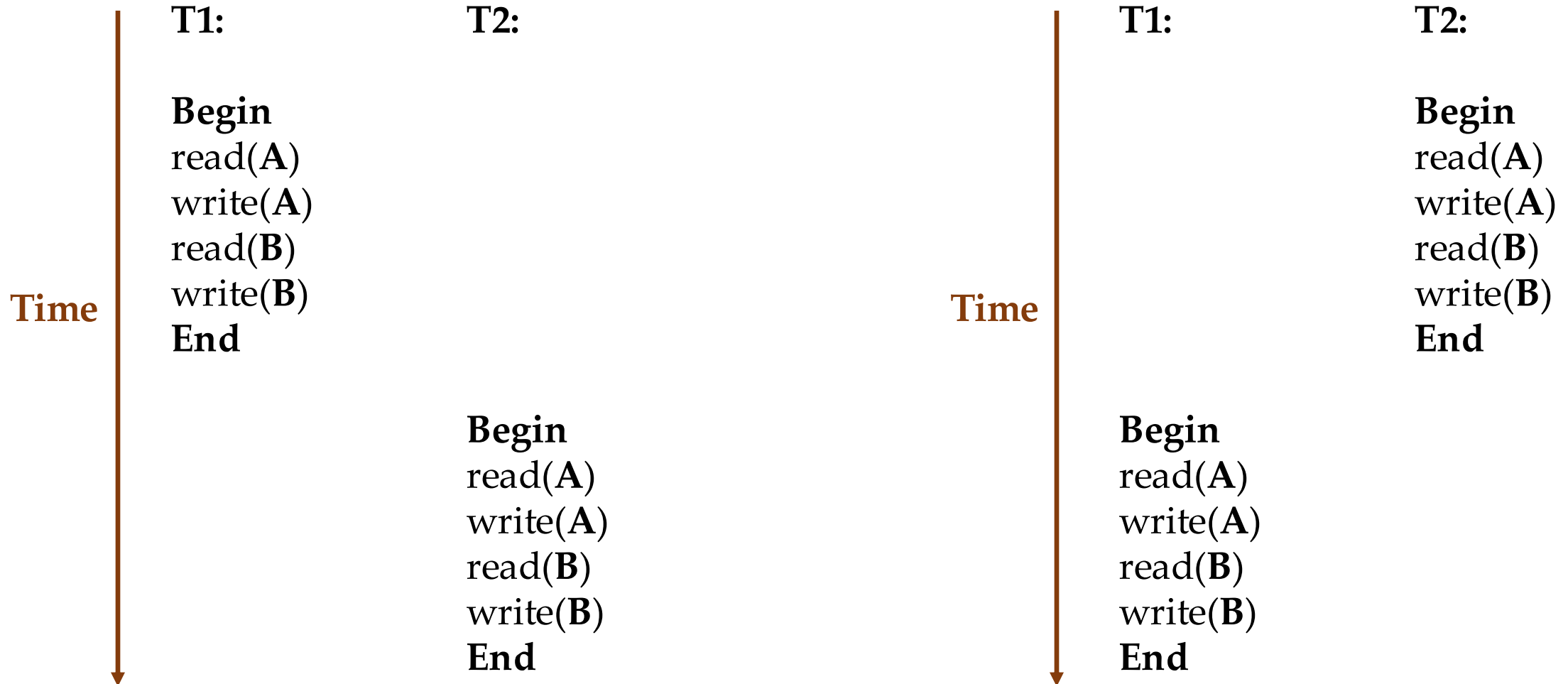
write(B)

**End**

# Serial Execution

- One legal interleaving is **serial execution**:
- Either  $T1 \rightarrow T2$ , or  $T2 \rightarrow T1$ .
  - Here, the notation  $T1 \rightarrow T2$  states that first execute transaction T1, and then execute T2.

# Serial Execution



# Serial Execution

- **Serial execution** is a legal interleaving:
- It guarantees **isolation**.
- But, serial execution does not take advantage of multi-core architecture.

# Schedule

- A **schedule** states the **order of executing** different operations of a transaction.
- The following is a **serial schedule** as it **does not interleave** the operations of different transactions.

Time

T1:

**Begin**

read(A)

write(A)

read(B)

write(B)

**End**

T2:

**Begin**

read(A)

write(A)

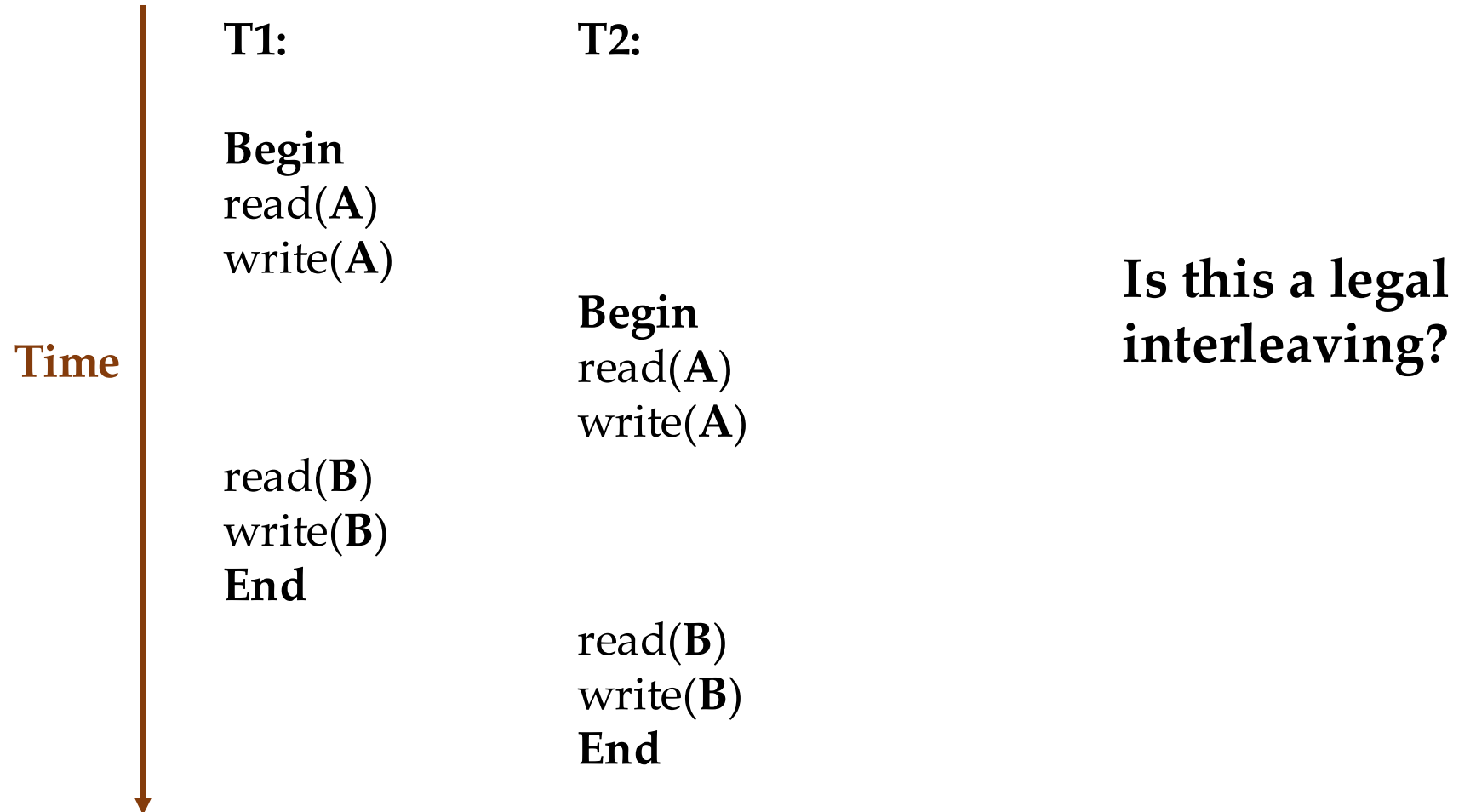
read(B)

write(B)

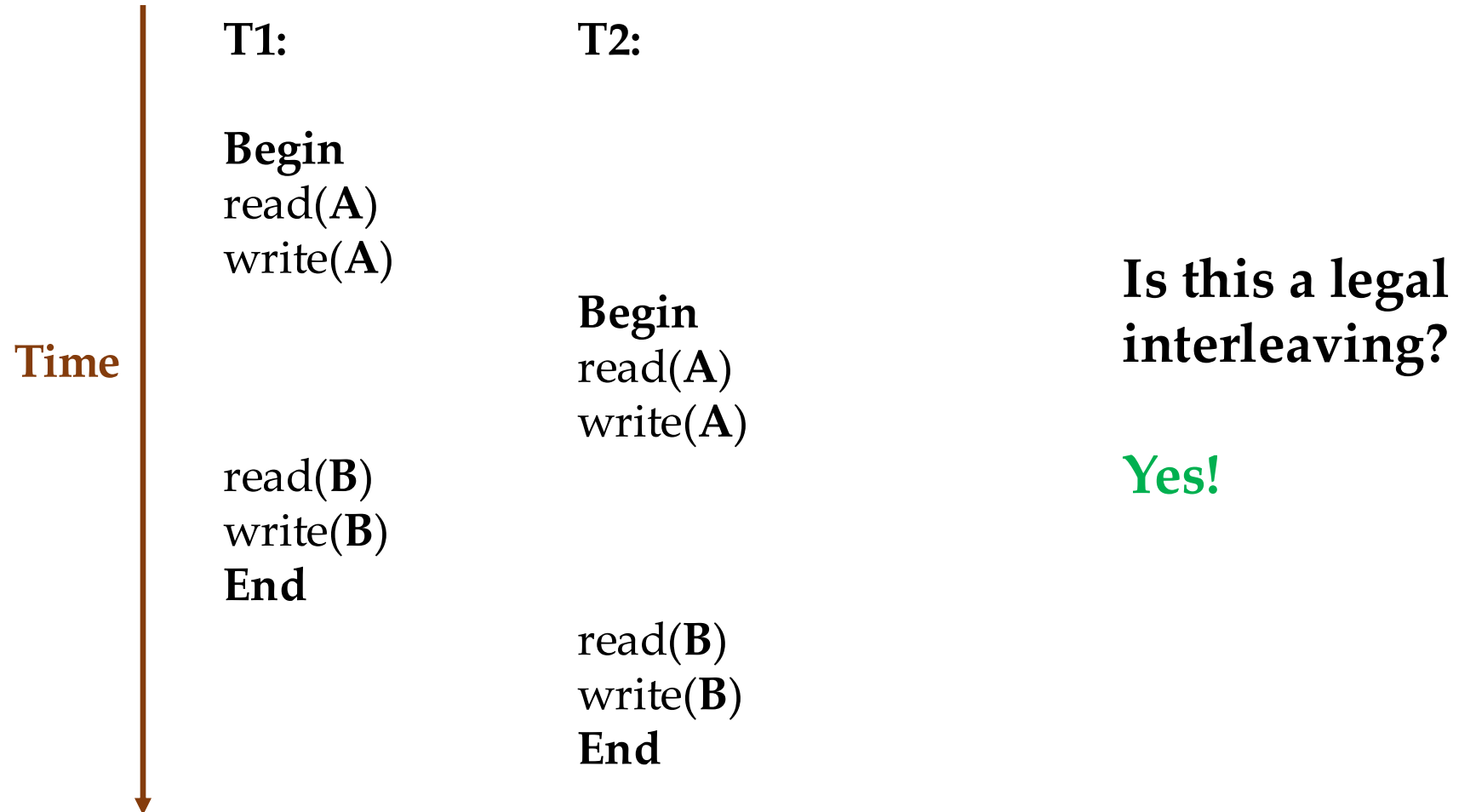
**End**



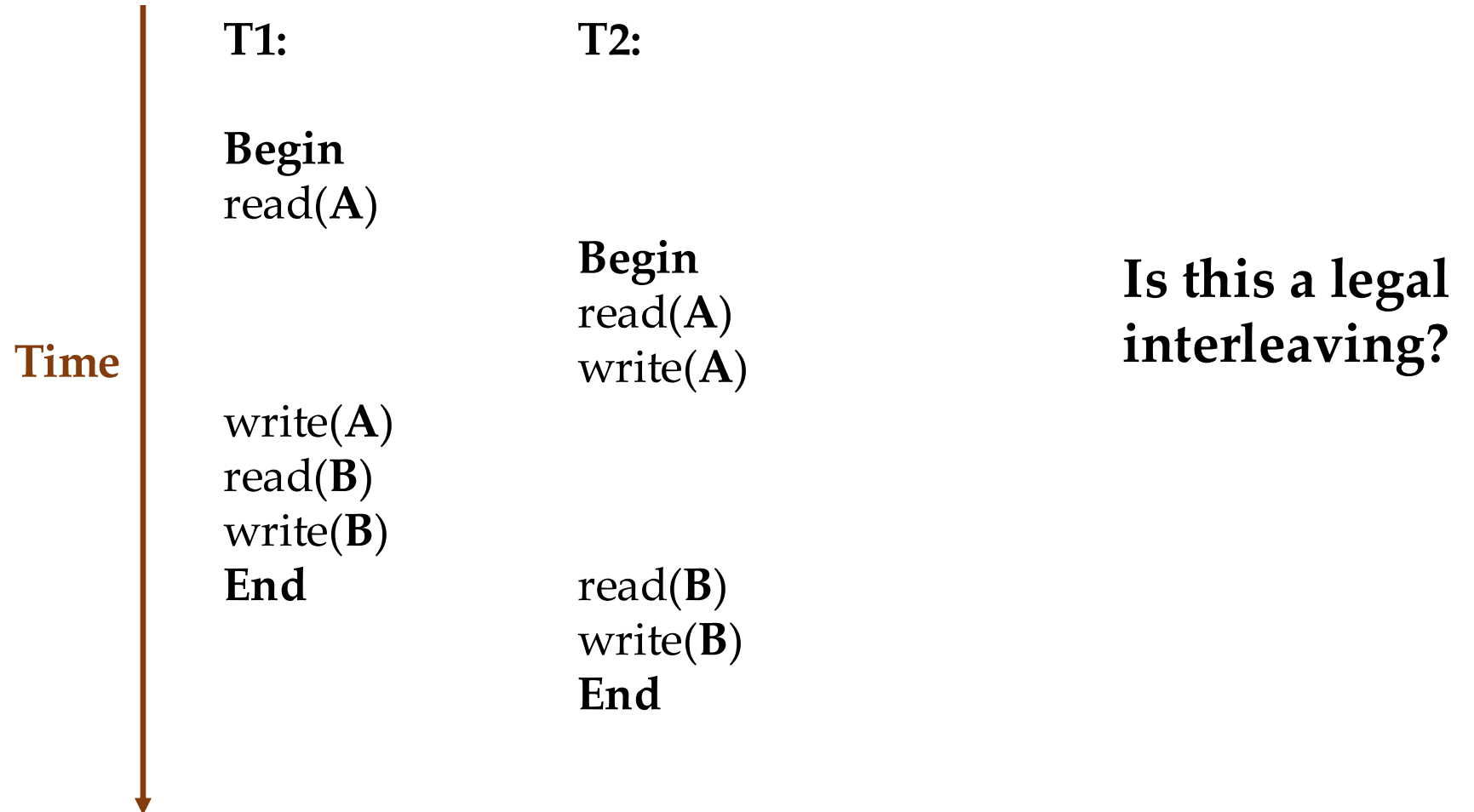
# Another Interleaving (I)



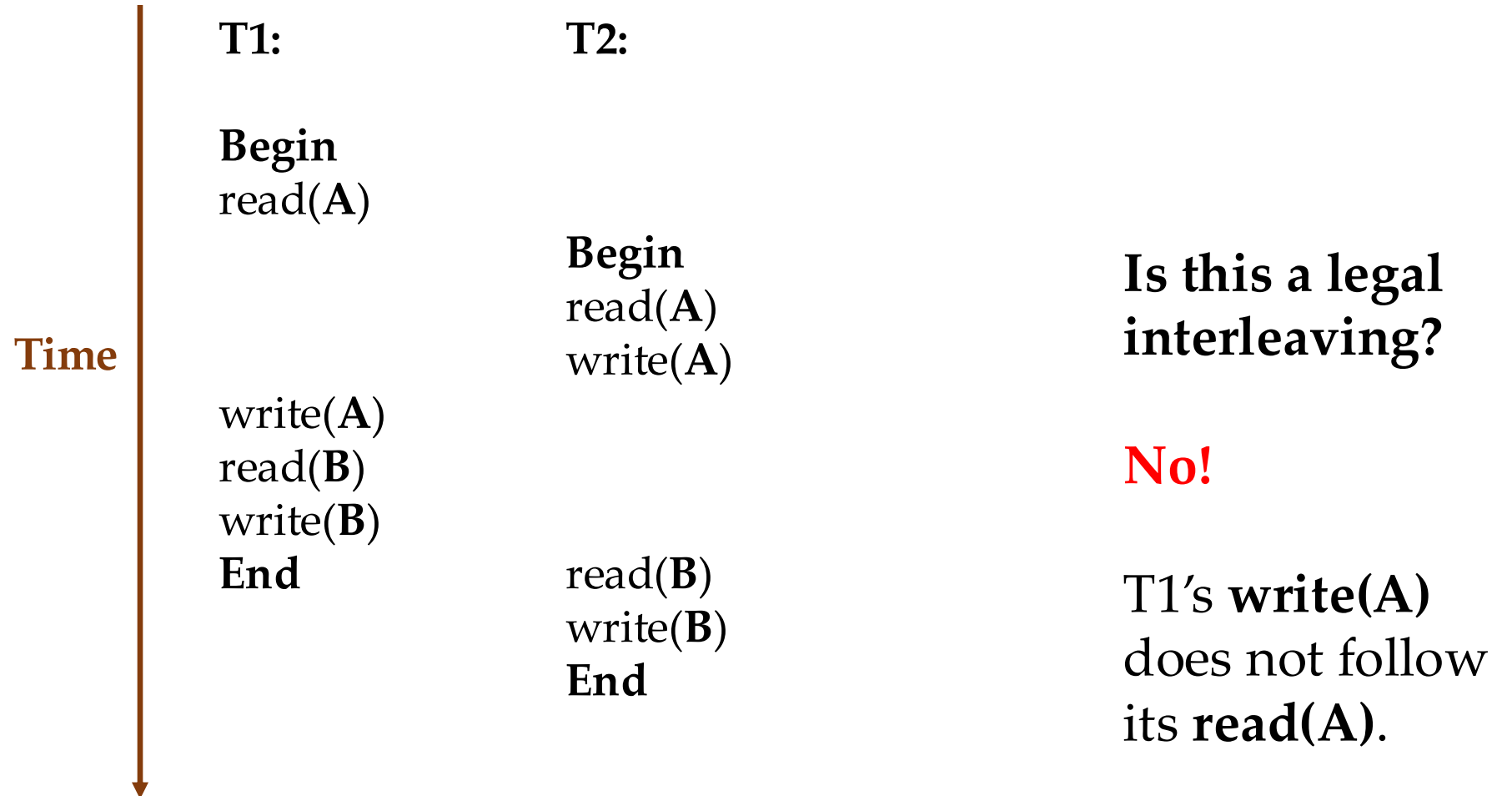
# Another Interleaving (I)



# Another Interleaving (II)



# Another Interleaving (II)



# Conflicting Transactions

Two transactions T1 and T2 if they **concurrently access the same variable** and at least one of that access is a **write** operation, then they **conflict**!

# Conflicting Transactions

- Interleaving concurrent transactions can lead to the following three anomalies:
  - Read-Write Conflicts (**R-W**)
  - Write-Read Conflicts (**W-R**)
  - Write-Write Conflicts (**W-W**)

# Read-Write Conflict

- Unrepeatable Read?

# Read-Write Conflict

- **Unrepeatable Read:** A transaction gets different values when reading the same object multiple times.
- Say, initially  $A = 50$ .

**T1:**

**Begin**  
read(A)

read(A)  
**End**

**T2:**

**Begin**  
read(A)  
write(A)  
**End**



# Read-Write Conflict

- **Unrepeatable Read:** A transaction gets different values when reading the same object multiple times.
- Say, initially  $A = 50$ .

Output → 50

T1:

Begin

read(A)

read(A)

End

T2:

Begin

read(A)

write(A)

End

# Read-Write Conflict

- **Unrepeatable Read:** A transaction gets different values when reading the same object multiple times.
- Say, initially  $A = 50$ .

Output → 50

**T1:**

**Begin**  
read(A)

read(A)  
**End**

**T2:**

**Begin**  
read(A)  
write(A)  
**End**

# Read-Write Conflict

- **Unrepeatable Read:** A transaction gets different values when reading the same object multiple times.
- Say, initially  $A = 50$ .

$A = 100$

**T1:**

**Begin**  
read(A)

read(A)  
**End**

**T2:**

**Begin**  
read(A)  
write(A)  
**End**

# Read-Write Conflict

- **Unrepeatable Read:** A transaction gets different values when reading the same object multiple times.
- Say, initially  $A = 50$ .

Output → 100

**T1:**

**Begin**  
read(A)

read(A)

**End**

**T2:**

**Begin**  
read(A)  
write(A)  
**End**

# Write-Read Conflict

- Dirty Read?

# Write-Read Conflict

- **Dirty Read:** One transaction reads data written by another transaction that has not committed yet.
- Say, initially  $A = 50$ .

**T1:**

**Begin**

read(A)

write(A)

**Abort**

**T2:**

**Begin**

read(A)

write(A)

**End**

# Write-Read Conflict

- **Dirty Read:** One transaction reads data written by another transaction that has not committed yet.
- Say, initially  $A = 50$ .

Output → 50

**T1:**

**Begin**

read(A)

write(A)

**Abort**

**T2:**

**Begin**

read(A)

write(A)

**End**

# Write-Read Conflict

- **Dirty Read:** One transaction reads data written by another transaction that has not committed yet.
- Say, initially  $A = 50$ .

$A = 100$

**T1:**

**Begin**

read(A)

write(A)

**Abort**

**T2:**

**Begin**

read(A)

write(A)

**End**



# Write-Read Conflict

- **Dirty Read:** One transaction reads data written by another transaction that has not committed yet.
- Say, initially  $A = 50$ .

**Output → 100**

**T1:**

**Begin**  
read(A)  
write(A)

**Abort**

**T2:**

**Begin**  
read(A)  
write(A)  
**End**

# Write-Read Conflict

- **Dirty Read:** One transaction reads data written by another transaction that has not committed yet.
- Say, initially  $A = 50$ .

**Output → 150**

**T1:**

**Begin**  
read(A)  
write(A)

**Abort**

**T2:**

**Begin**  
read(A)  
write(A)  
**End**

# Write-Read Conflict

- **Dirty Read:** One transaction reads data written by another transaction that has not committed yet.
- Say, initially  $A = 50$ .

**Transaction T1  
has to be aborted**

**T1:**

**Begin**  
read(A)  
write(A)

**Abort**

**T2:**

**Begin**  
read(A)  
write(A)  
**End**

# Write-Write Conflict

- Lost Update?

# Write-Write Conflict

- **Lost Update** → One transaction overwrites uncommitted data from another uncommitted transaction.
- Say, initially  $A = 50$ .

**T1:**

**Begin**  
read(A)

write(A)

write(B)  
**End**

**T2:**

**Begin**  
read(A)

write(A)  
write(B)  
**End**

# Write-Write Conflict

- **Lost Update** → One transaction overwrites uncommitted data from another uncommitted transaction.
- Say, initially  $A = 50$ .

**Output → 50**

**T1:**

**Begin**

read(A)

write(A)

write(B)

**End**

**T2:**

**Begin**

read(A)

write(A)

write(B)

**End**

# Write-Write Conflict

- **Lost Update** → One transaction overwrites uncommitted data from another uncommitted transaction.
- Say, initially  $A = 50$ .

**Output → 50**

**T1:**

**Begin**  
read(A)

write(A)

write(B)  
**End**

**T2:**

**Begin**  
read(A)

write(A)  
write(B)  
**End**

# Write-Write Conflict

- **Lost Update** → One transaction overwrites uncommitted data from another uncommitted transaction.
- Say, initially  $A = 50$ .

$A = 100$

**T1:**

**Begin**  
read(A)

write(A)

write(B)  
**End**

**T2:**

**Begin**  
read(A)

write(A)  
write(B)  
**End**



# Write-Write Conflict

- **Lost Update** → One transaction overwrites uncommitted data from another uncommitted transaction.
- Say, initially  $A = 50$ .

**$A = 100$**

Previous update  
missed!

**T1:**

**Begin**  
read(A)

write(A)

write(B)  
**End**

**T2:**

**Begin**  
read(A)

**write(A)**  
write(B)  
**End**

# Write-Write Conflict

- **Lost Update** → One transaction overwrites uncommitted data from another uncommitted transaction.
- Say, initially  $A = 50$ .

**A = 100**

**B = 100**

**T1:**

**Begin**  
read(A)

write(A)

write(B)  
**End**

**T2:**

**Begin**  
read(A)

write(A)

write(B)

**End**

# Write-Write Conflict

- **Lost Update** → One transaction overwrites uncommitted data from another uncommitted transaction.
- Say, initially  $A = 50$ .

**$A = 100$**

**$B = 150$**

**T1:**

**Begin**  
read(A)

write(A)

write(B)

**End**

**T2:**

**Begin**  
read(A)

write(A)

write(B)

**End**

# Write-Write Conflict

- **Lost Update** → One transaction overwrites uncommitted data from another uncommitted transaction.
- Say, initially  $A = 50$ .

**A = 100**

**B = 150**

**T1:**

**Begin**  
read(A)

write(A)

write(B)

**End**

**T2:**

**Begin**  
read(A)

write(A)  
write(B)  
**End**

This leads to unexpected results of 100,150 when it should have been 150, 150.

# Isolation Support: Concurrency Control

- A **concurrency control protocol** lays down the mechanism for the DBMS to decide a legal/valid schedule of transactions.
- What are the **types** of concurrency control protocols?

# Isolation Support: Concurrency Control

- A concurrency control protocol lays down the mechanism for the DBMS to decide legal/valid schedule of transactions.
- What are the types of concurrency control protocols?
- **Pessimistic:** Prevent problems from arising in the first place.

# Isolation Support: Concurrency Control

- A concurrency control protocol lays down the mechanism for the DBMS to decide legal/valid schedule of transactions.
- What are the types of concurrency control protocols?
- **Pessimistic:** Prevent problems from arising in the first place.
- **Optimistic:** Assume that conflicts are rare; deal with them after they occur.

# Isolation Support: Concurrency Control

- A concurrency control protocol lays down the mechanism for the DBMS to decide legal/valid schedule of transactions.
- What are the types of concurrency control protocols?
- **Pessimistic:** Prevent problems from arising in the first place.
- **Optimistic:** Assume that conflicts are rare; deal with them after they occur.
- **But, how does a concurrency control protocol determine a valid schedule?**



# Serializable Schedules

- **Serial Schedule** → A schedule that does not interleave the operations of different transactions.
- **Equivalent Schedule** → For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.
- **Serializable Schedule?**

# Serializable Schedules

- **Serial Schedule** → A schedule that does not interleave the operations of different transactions.
- **Equivalent Schedule** → For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.
- **Serializable Schedule** → A schedule that is equivalent to some serial execution of the transactions (serial schedule).

# Serializable Schedules

- **Serial Schedule** → A schedule that does not interleave the operations of different transactions.
- **Equivalent Schedule** → For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.
- **Serializable Schedule** → A schedule that is equivalent to some serial execution of the transactions (serial schedule).
- If each transaction preserves consistency, then the corresponding serializable schedule preserves consistency!

# Isolation Levels vs. Consistency Levels

Isolation Levels	Consistency Levels
------------------	--------------------

# Isolation Levels vs. Consistency Levels

Isolation Levels	Consistency Levels
<ul style="list-style-type: none"><li>• <b>Correspond</b> to the <b>I</b> in <b>ACID</b>.</li><li>• <b>Database isolation</b> is the ability of a database to allow a transaction to execute as if there are no other concurrently running transactions.</li><li>• Greater the guaranteed isolation among the transactions, lesser the system performance.</li><li>• <b>Isolation levels</b> trade off isolation guarantees for improved performance.</li></ul>	

# Isolation Levels vs. Consistency Levels

## Isolation Levels

- **Correspond** to the **I** in **ACID**.
- **Database isolation** is the ability of a database to allow a transaction to execute as if there are no other concurrently running transactions.
- Greater the guaranteed isolation among the transactions, lesser the system performance.
- **Isolation levels** trade off isolation guarantees for improved performance.

## Consistency Levels

- **Do not correspond** to **C** in **ACID**.
- Unlike the **C** in **ACID**, the **database consistency** refers to the rules that make a **concurrent, distributed system** appear as a **single-threaded, centralized system**.
- **Reads** at a particular point in time **must reflect the most recently completed write** (in real-time) of that data item, no matter which server processed that write.
- **Consistency levels** trade off read results for improved performance.

# Isolation Levels vs. Consistency Levels

- More simply said:
  - Whenever you talk about transaction isolation, you will be talking about isolation levels.
  - Whenever you talk about individual operations like read/write, you will talk about consistency levels.

# Serializable Isolation Level

- Also known as serializability.
- The ability of a DBMS to run transactions in parallel, but in a way that they are running, **serially**, that is, **one after another**.
- Thus, if the DBMS can ensure a serializable schedule for a set of transactions, then we say that the DBMS is offering serializability or serializable isolation level.



# Levels of Serializability

- **Conflict Serializability**
  - Most DBMS try to support this.
- **View Serializability**
  - No DBMS can do this!

# How to determine the Level of Serializability?

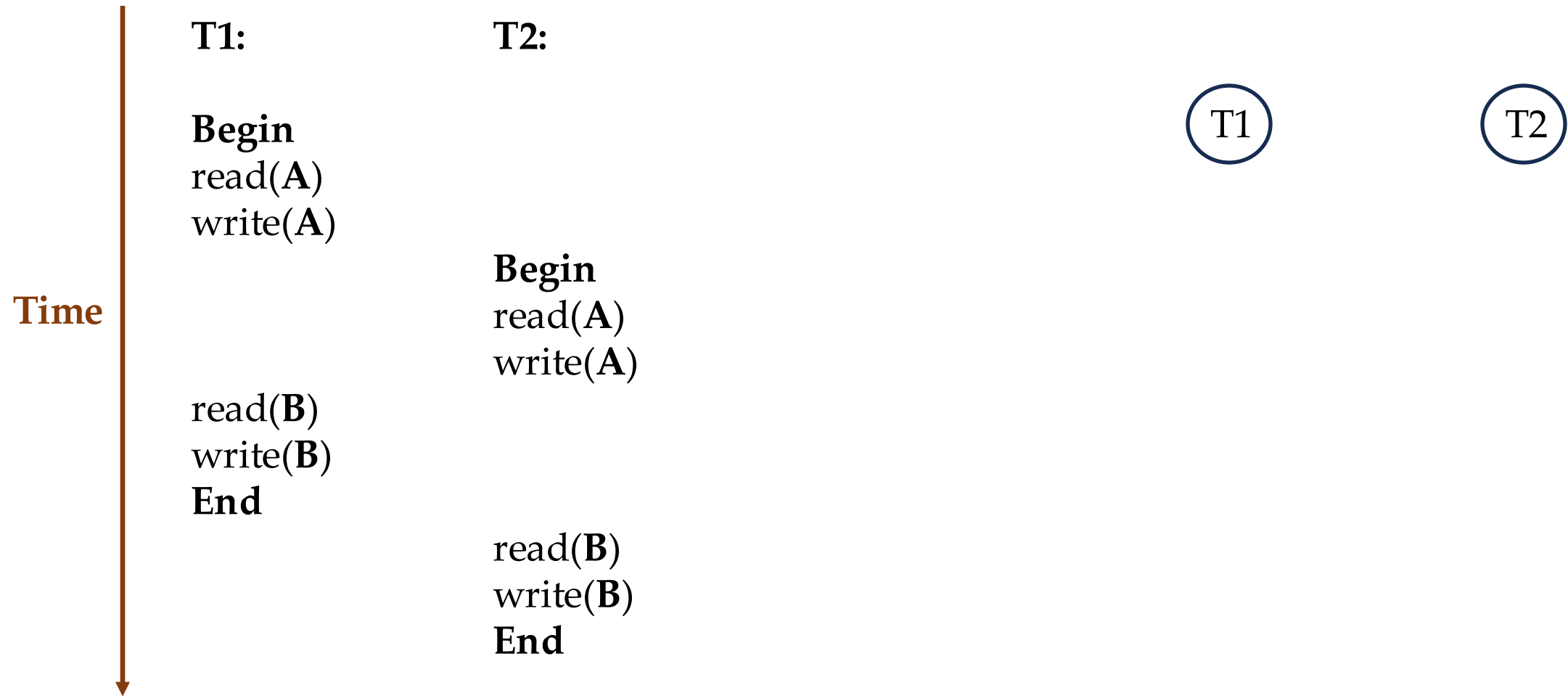
# How to determine the level of serializability?

- **Dependency Graph!**
- **How can you create a dependency graph?**

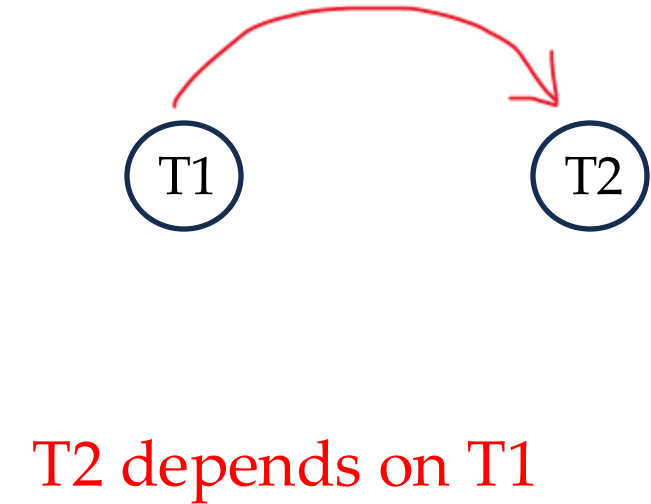
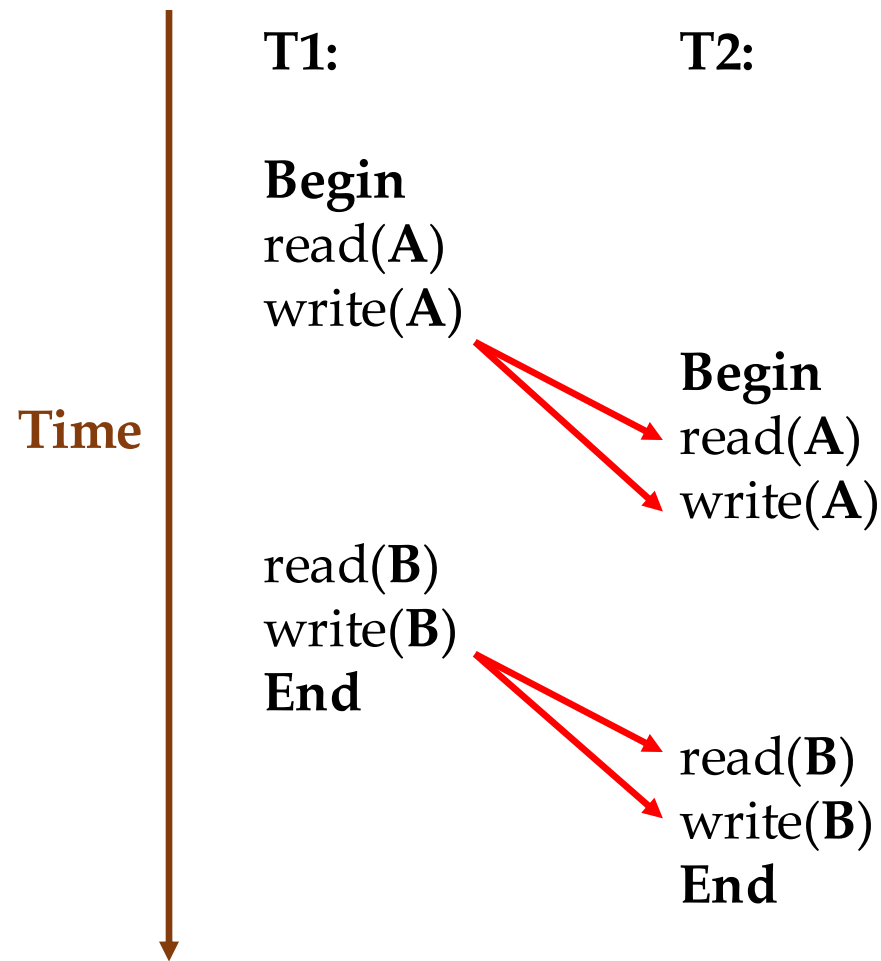
# Dependency Graphs

- Help to determine the level of serializability among other things.
- How can you create a dependency graph?
- One node per transaction.
- **Add an edge from transaction  $T_i$  to transaction  $T_j$  if you the following are met:**
  - An operation  **$O_i$  of  $T_i$**  conflicts with an operation  **$O_j$  of  $T_j$** .
  - **$O_i$**  appears earlier in the schedule than  **$O_j$** .
- Also known as **precedence graph**.

# Dependency Graphs Example I

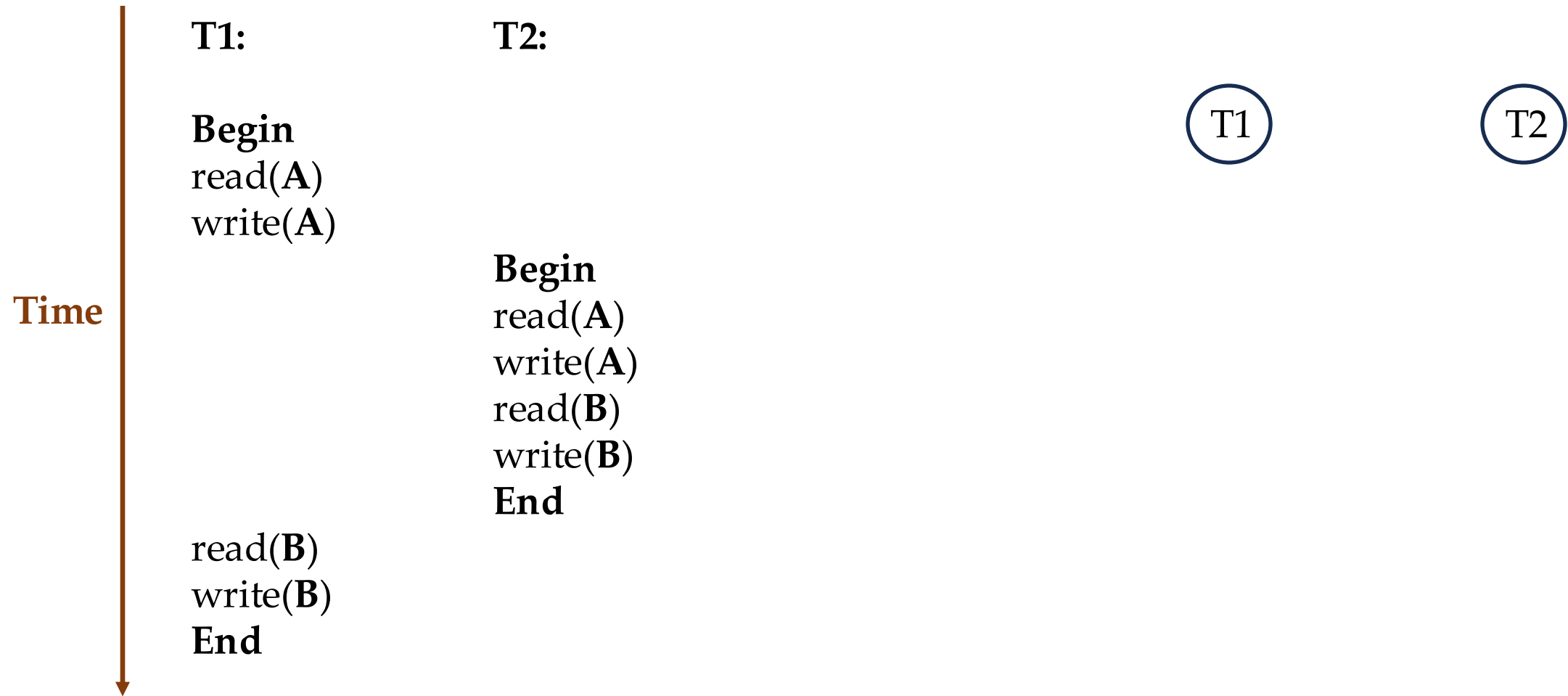


# Dependency Graphs Example I

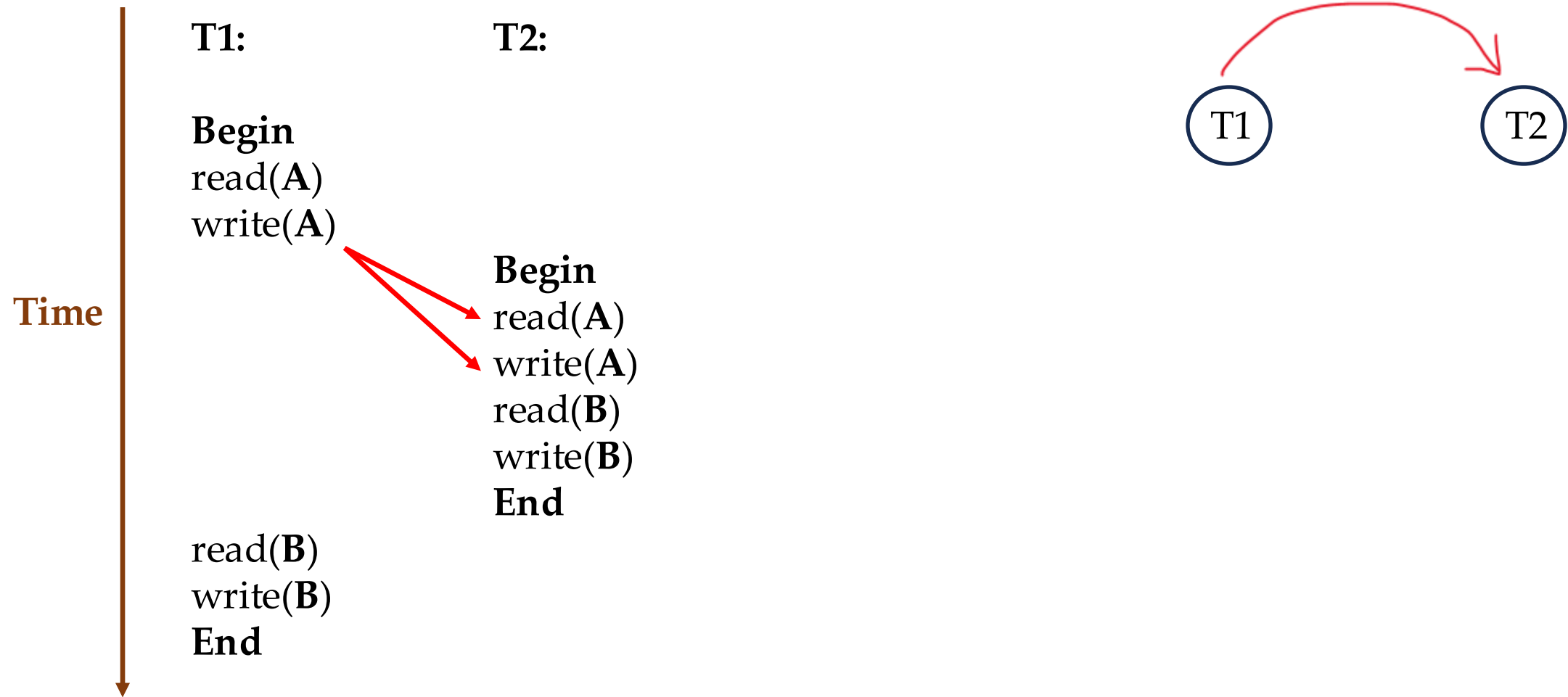


W-R and W-W Conflicts

# Dependency Graphs Example II



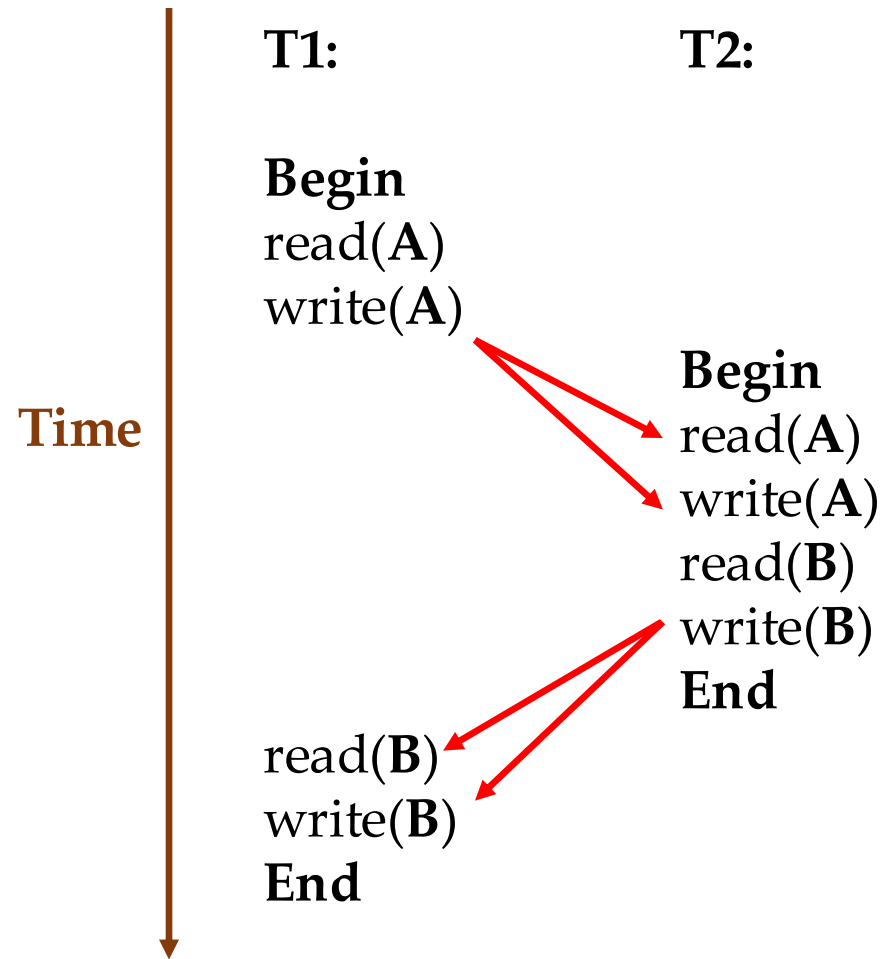
# Dependency Graphs Example II



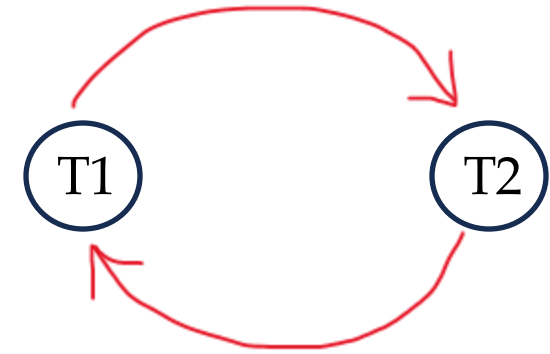
W-R and W-W Conflicts



# Dependency Graphs Example II



W-R and W-W Conflicts



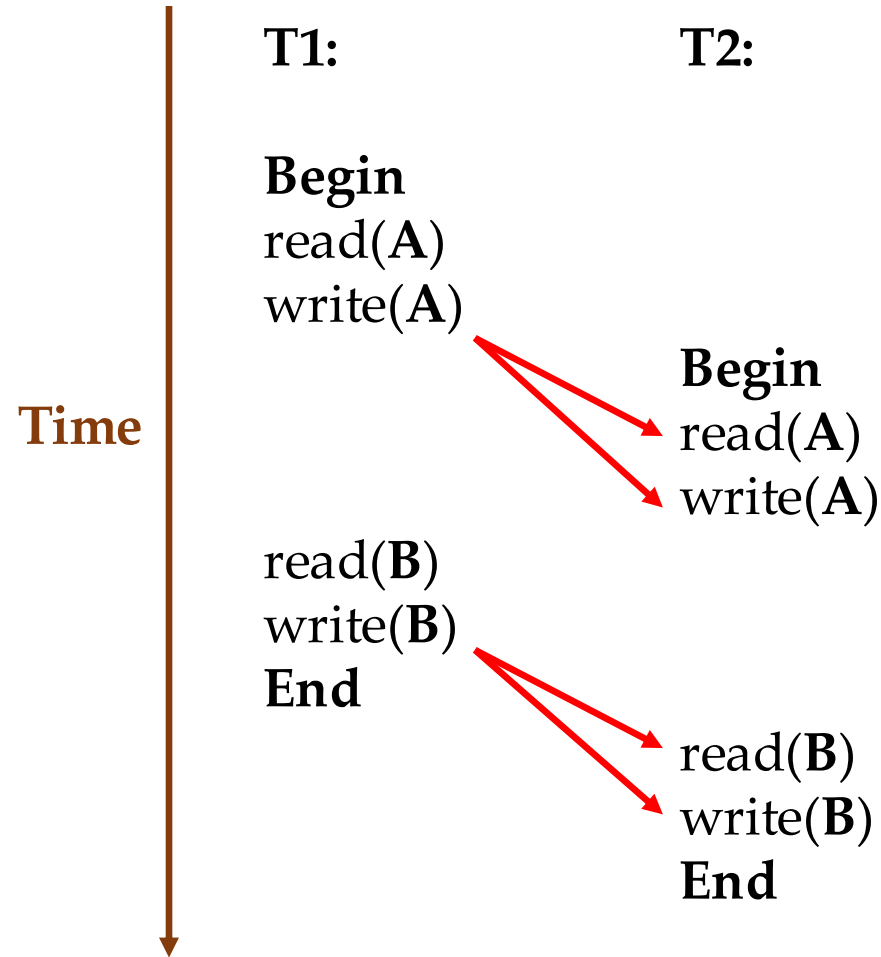
Conflict Cycle!

# Conflict Serializability

# Conflict Serializability

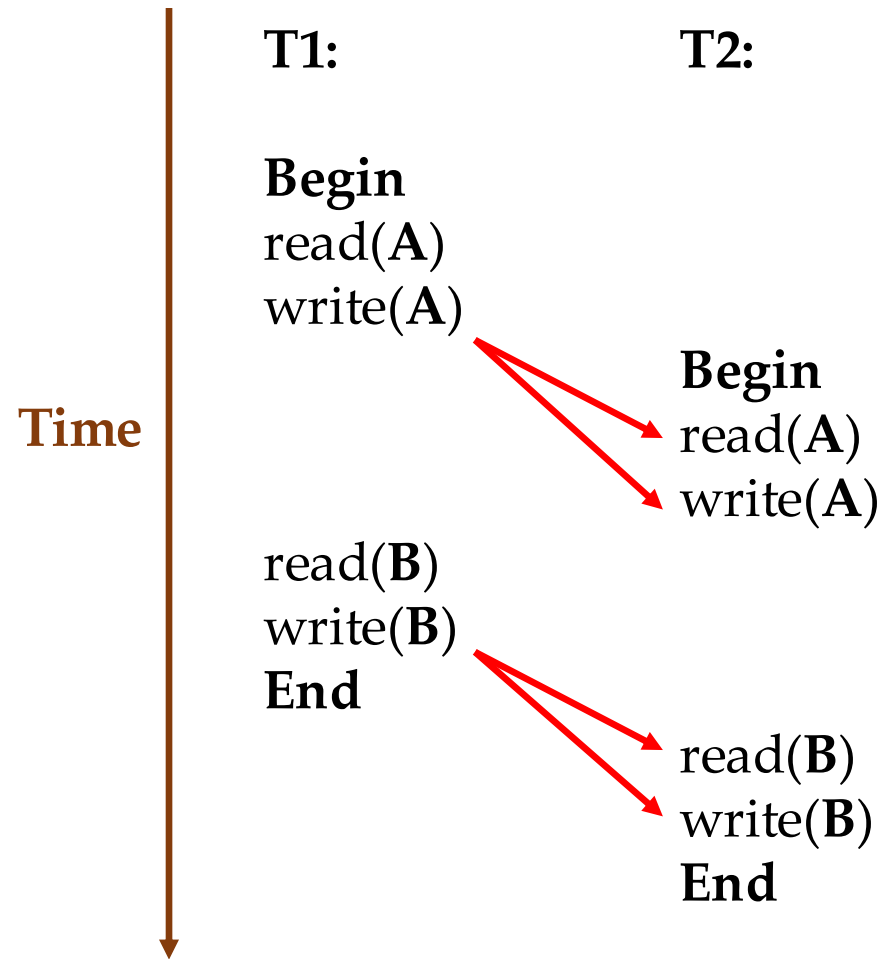
- Two schedules are **conflict equivalent** if and only if:
  - They involve the **same actions** of the same transactions.
  - Every pair of conflicting actions is ordered the same way.
- A schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule.
- You can **transform** a conflict serializable schedule S into a serial schedule by swapping consecutive non-conflicting operations of different transactions.
- A schedule is conflict serializable iff its **dependency graph is acyclic**.

# Conflict Serializability: Example I



Is this conflict serializable?

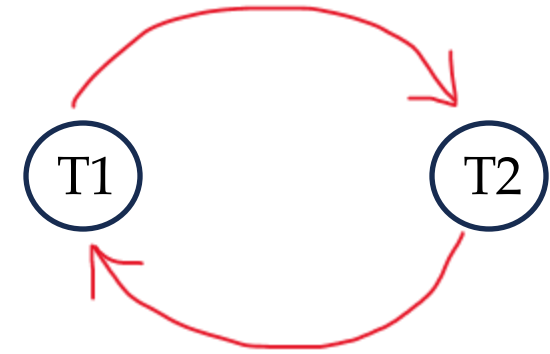
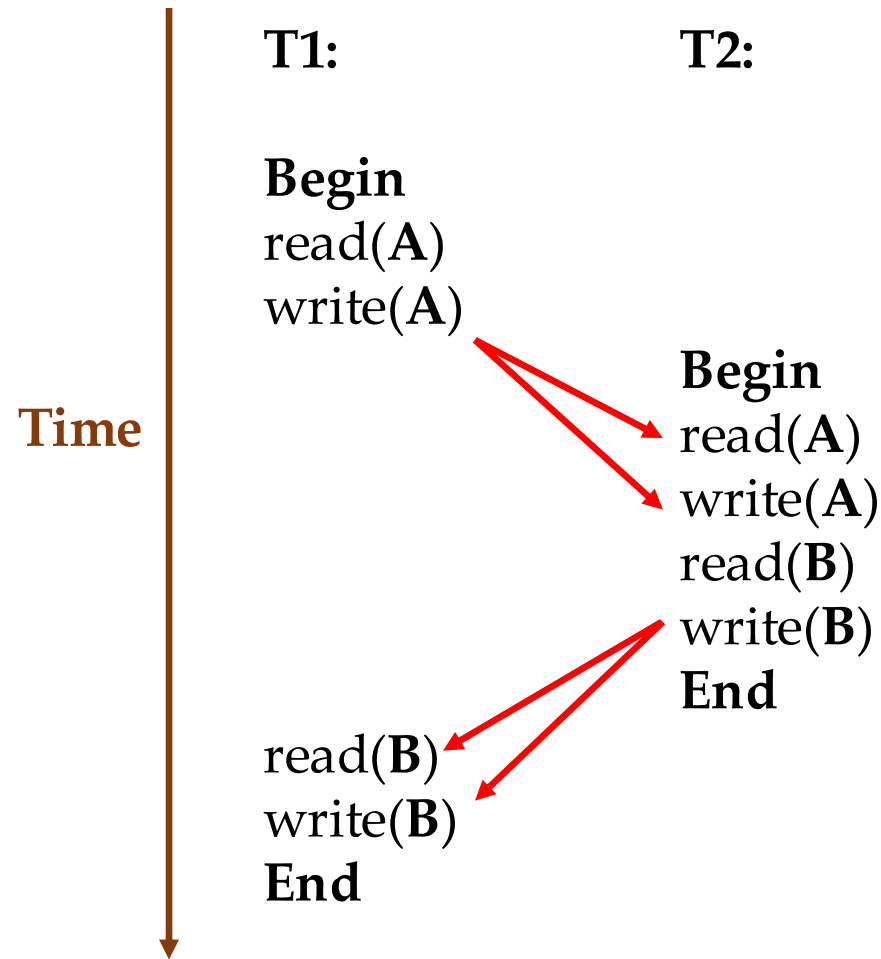
# Conflict Serializability: Example I



Is this conflict serializable?

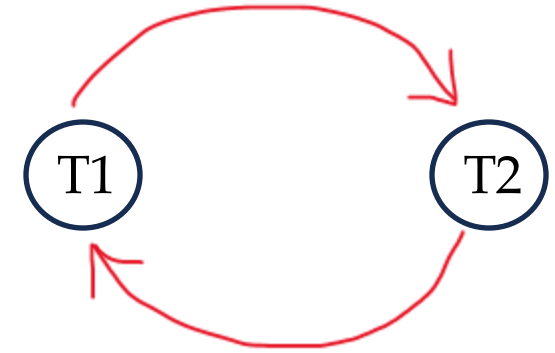
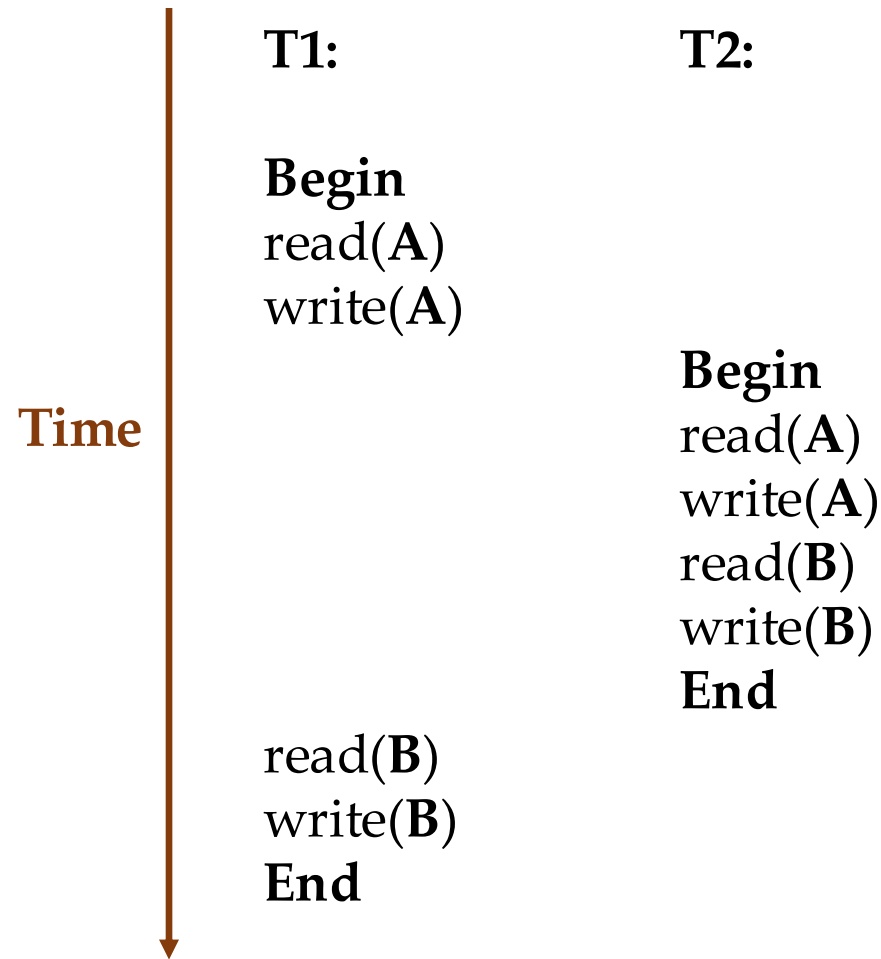
Yes!

# Conflict Serializability: Example II



Is this conflict serializable?

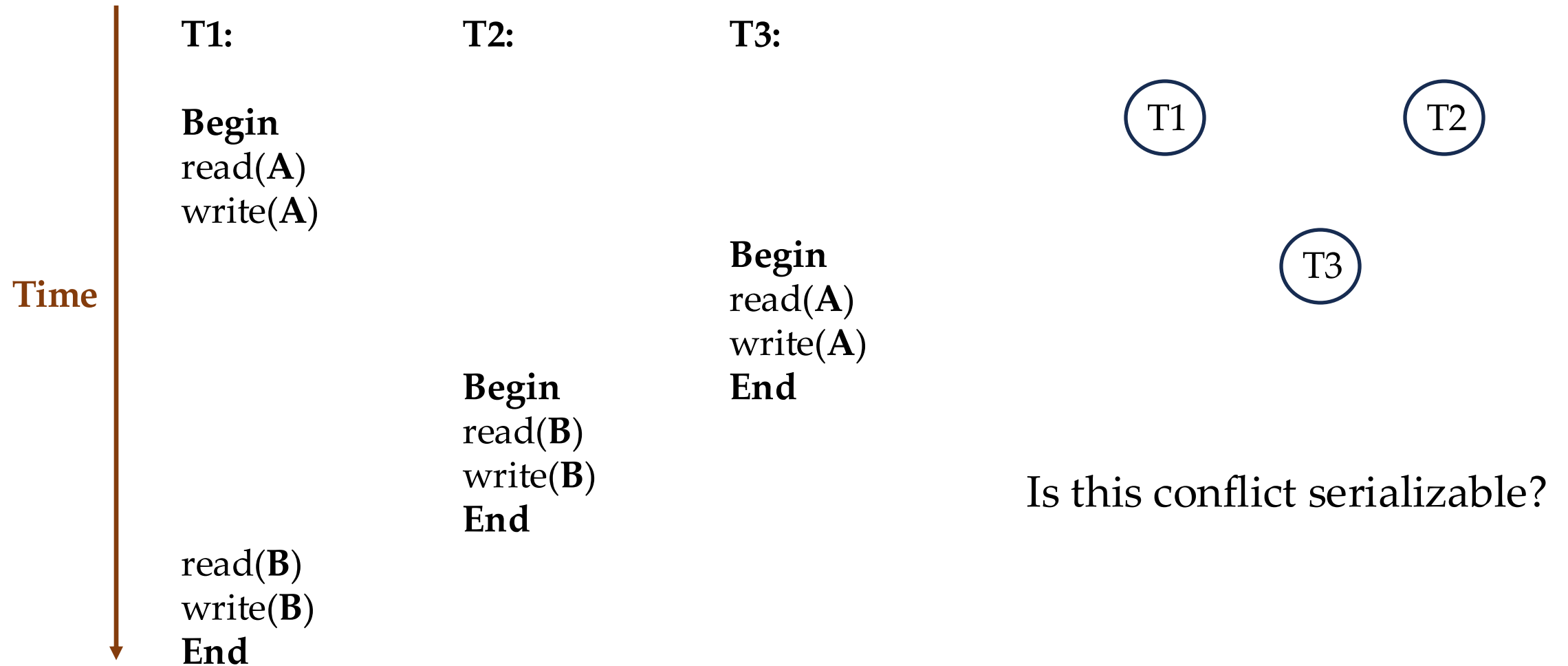
# Conflict Serializability: Example II



Is this conflict serializable?

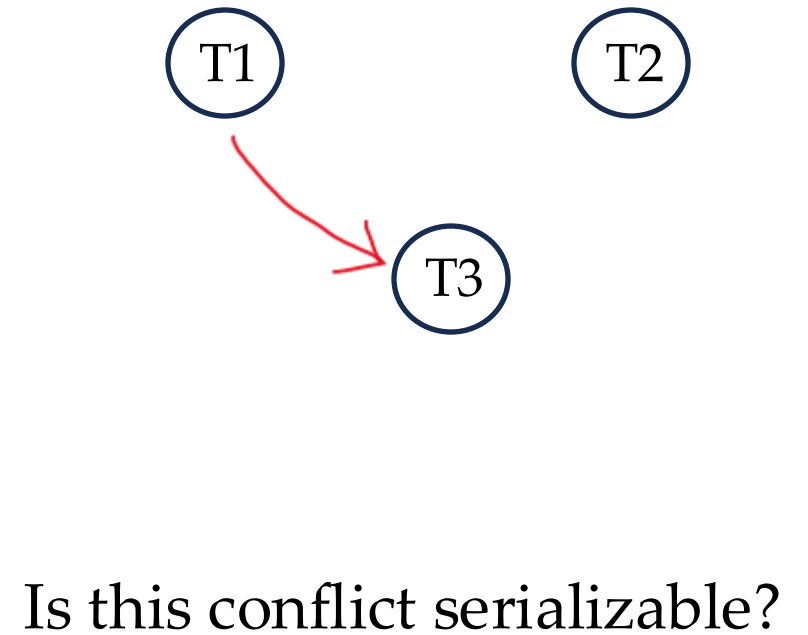
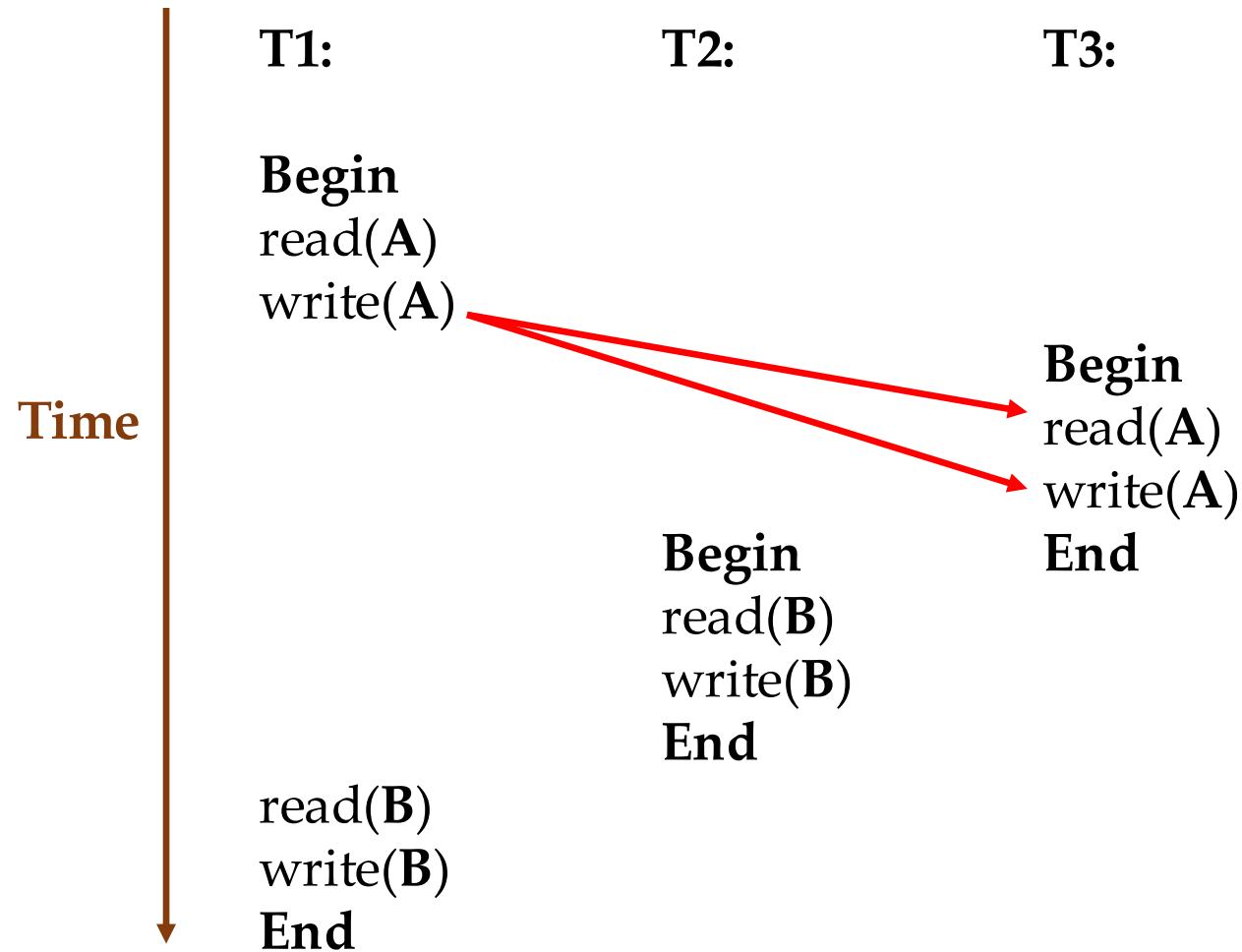
**No!**

# Conflict Serializability: Example III

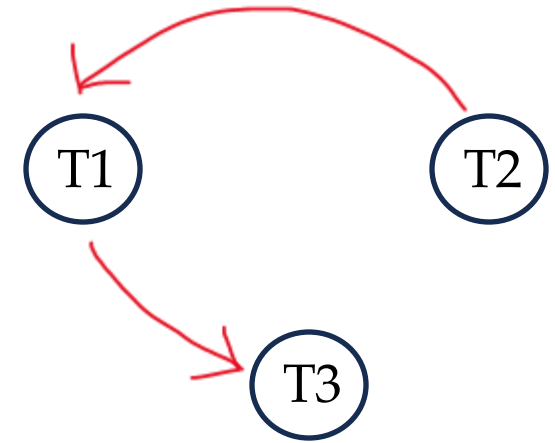
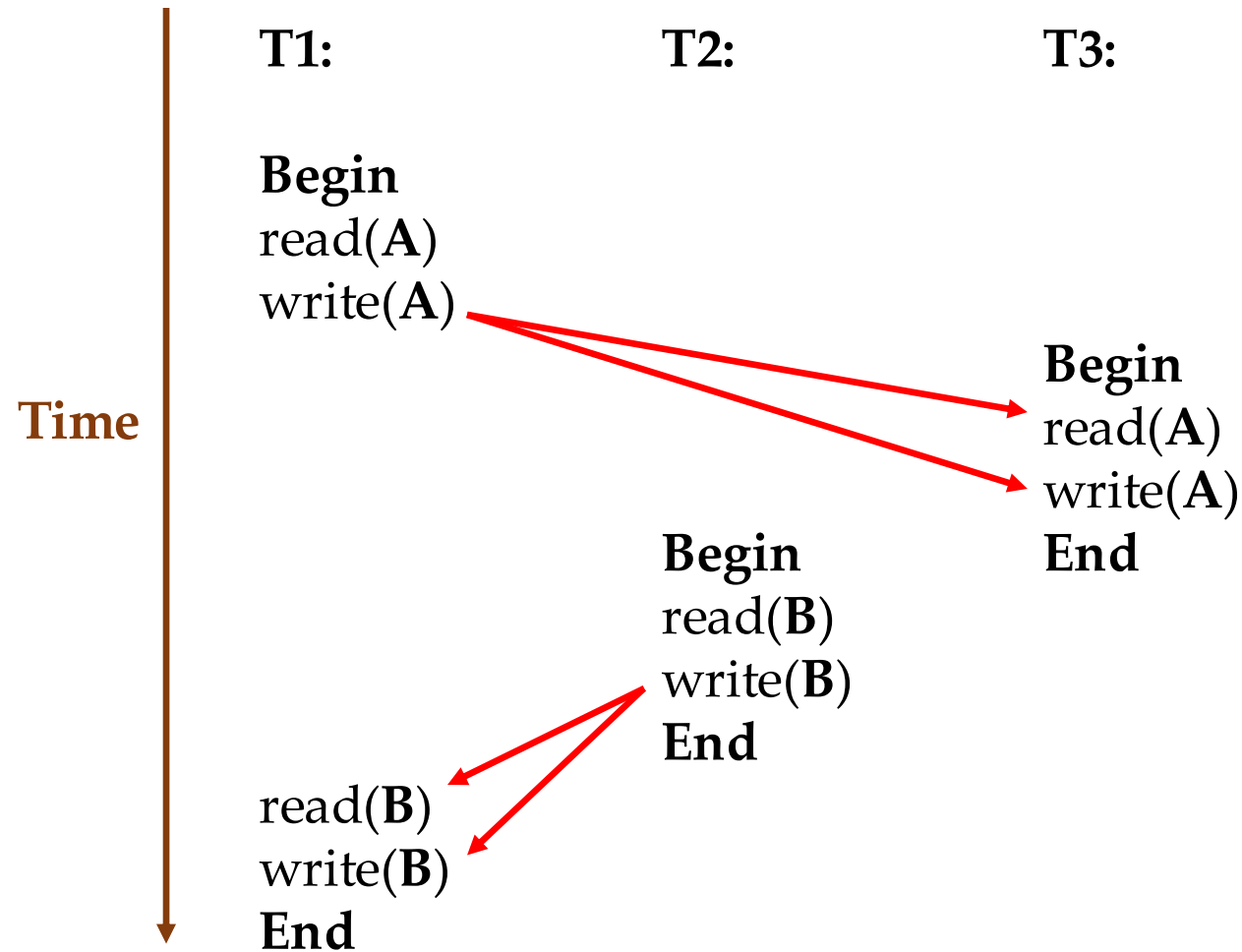




# Conflict Serializability: Example III

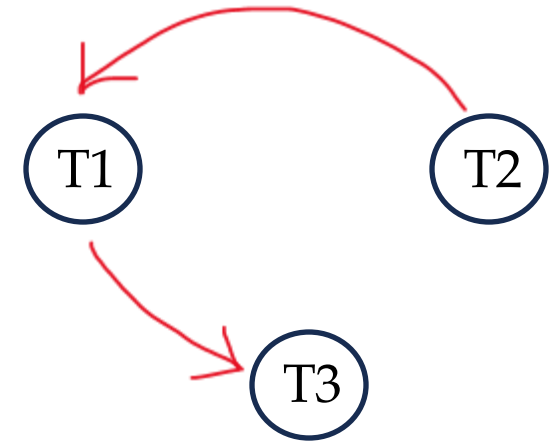
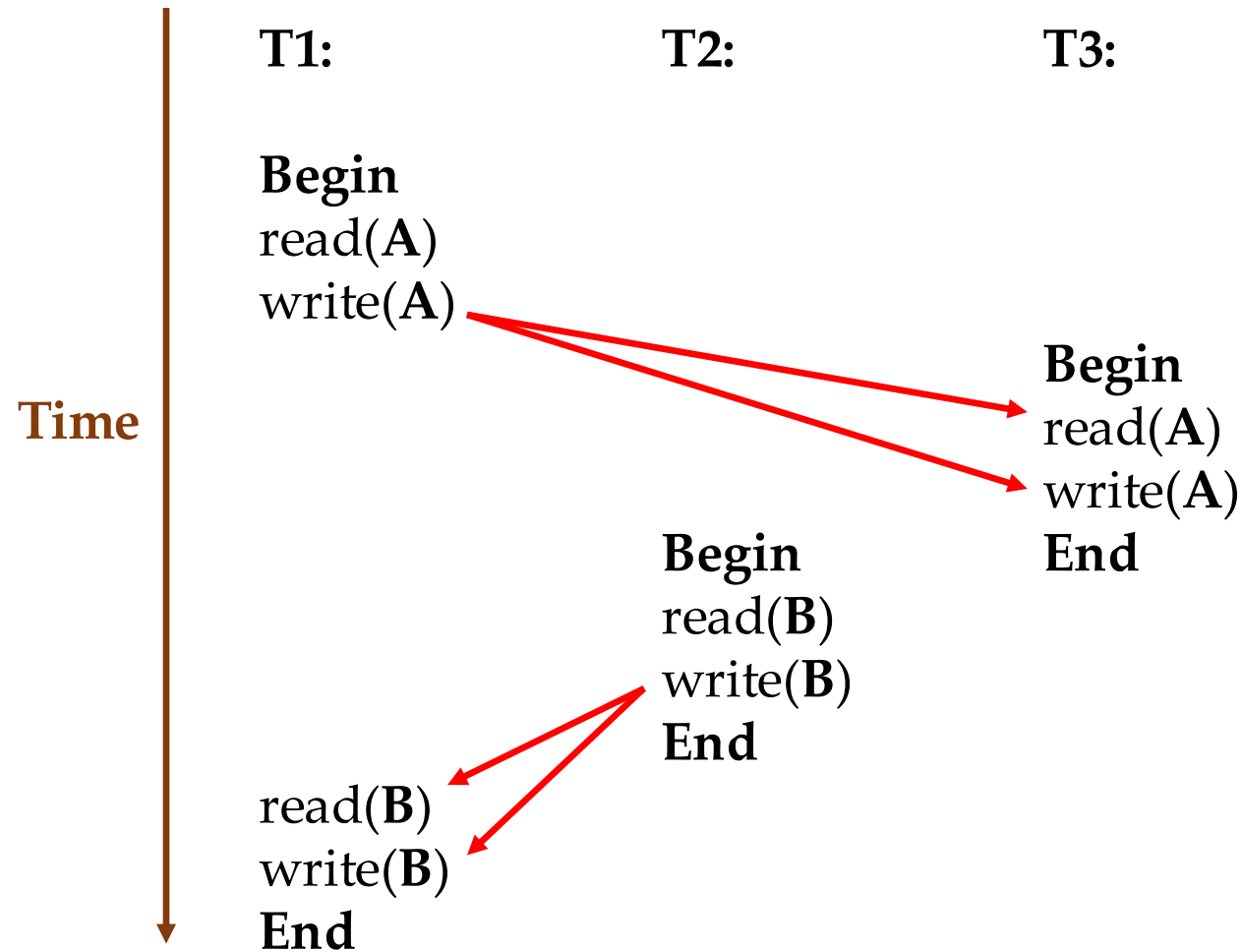


# Conflict Serializability: Example III



Is this conflict serializable?

# Conflict Serializability: Example III



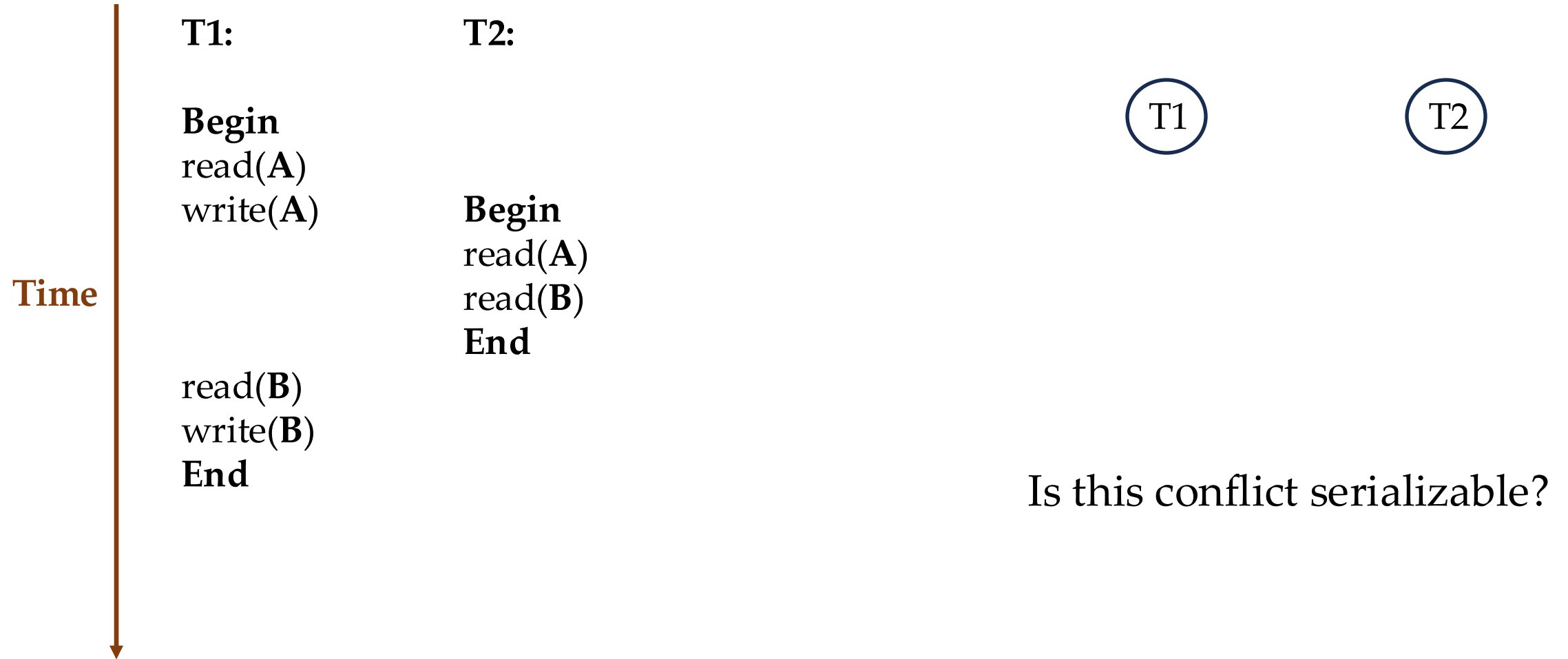
Is this conflict serializable?

Is this equivalent to a serial schedule?

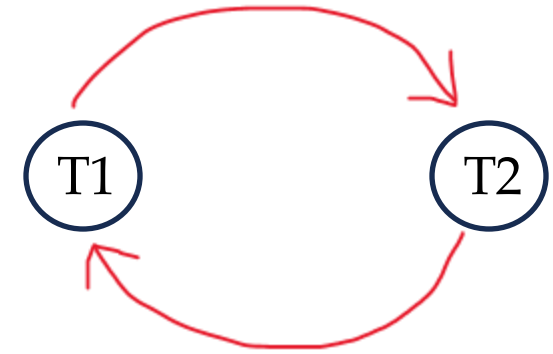
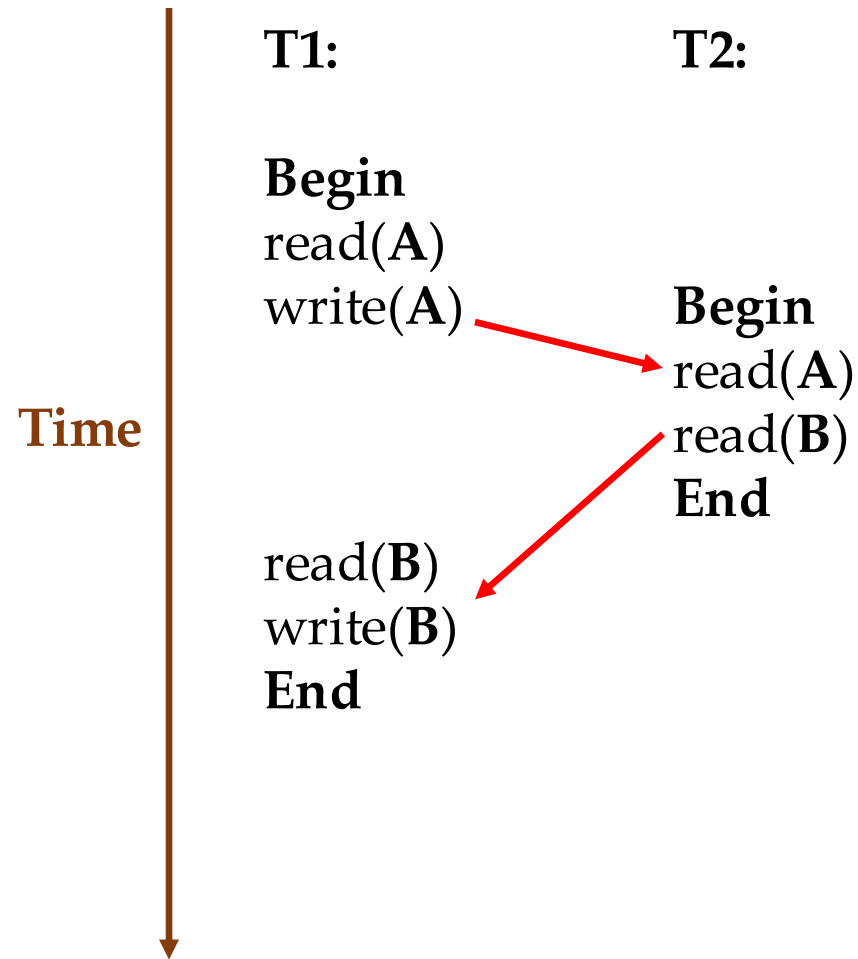
**Yes! {T2, T1, T3}**

**T2 should execute before T3!**

# Conflict Serializability: Example IV



# Conflict Serializability: Example IV



Is this conflict serializable?

No!

# Determining Serializability Order from Dependence Graph

# Determining Serializability Order from Dependence Graph

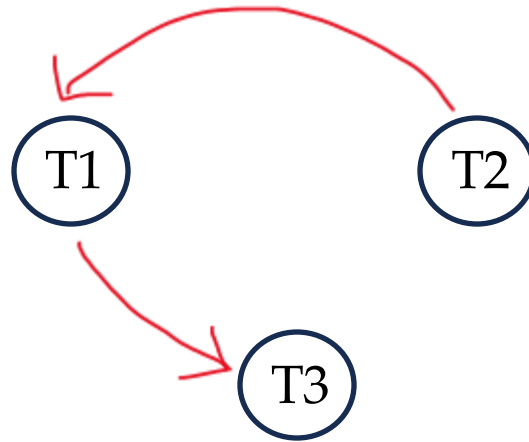
- Dependence graph only dictates the dependency between the transactions.
- To find the serializability order of these transactions, you run **topological sort**.
- Can there be multiple possible serializability orders?

# Determining Serializability Order from Dependence Graph

- Dependence graph only dictates the dependency between the transactions.
- To find the serializability order of these transactions, you run **topological sort**.
- Can there be multiple possible serializability orders?
  - **Yes!**
  - Not every pair of transactions are dependent on each other.

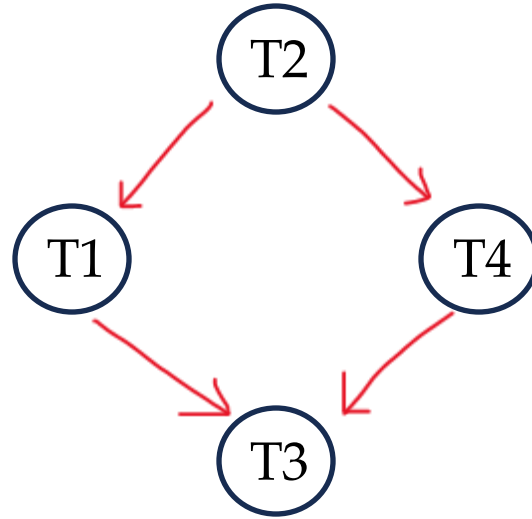


# Determining Serializability Order from Dependence Graph



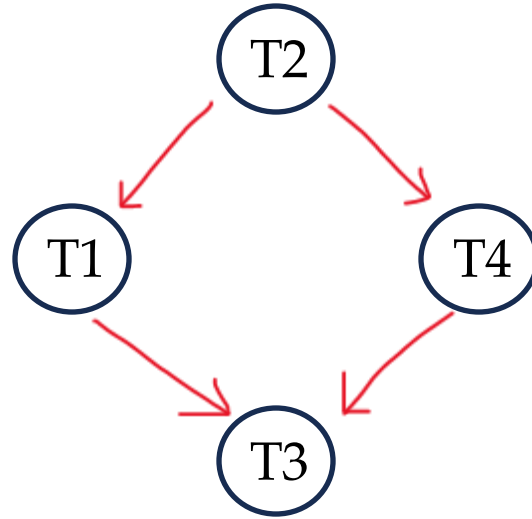
Serializability order: {T2, T1, T3}

# Determining Serializability Order from Dependence Graph



Serializability order?

# Determining Serializability Order from Dependence Graph



Serializability orders: {T2, T1, T4, T3} or  
{T2, T4, T1, T3}

# View Serializability

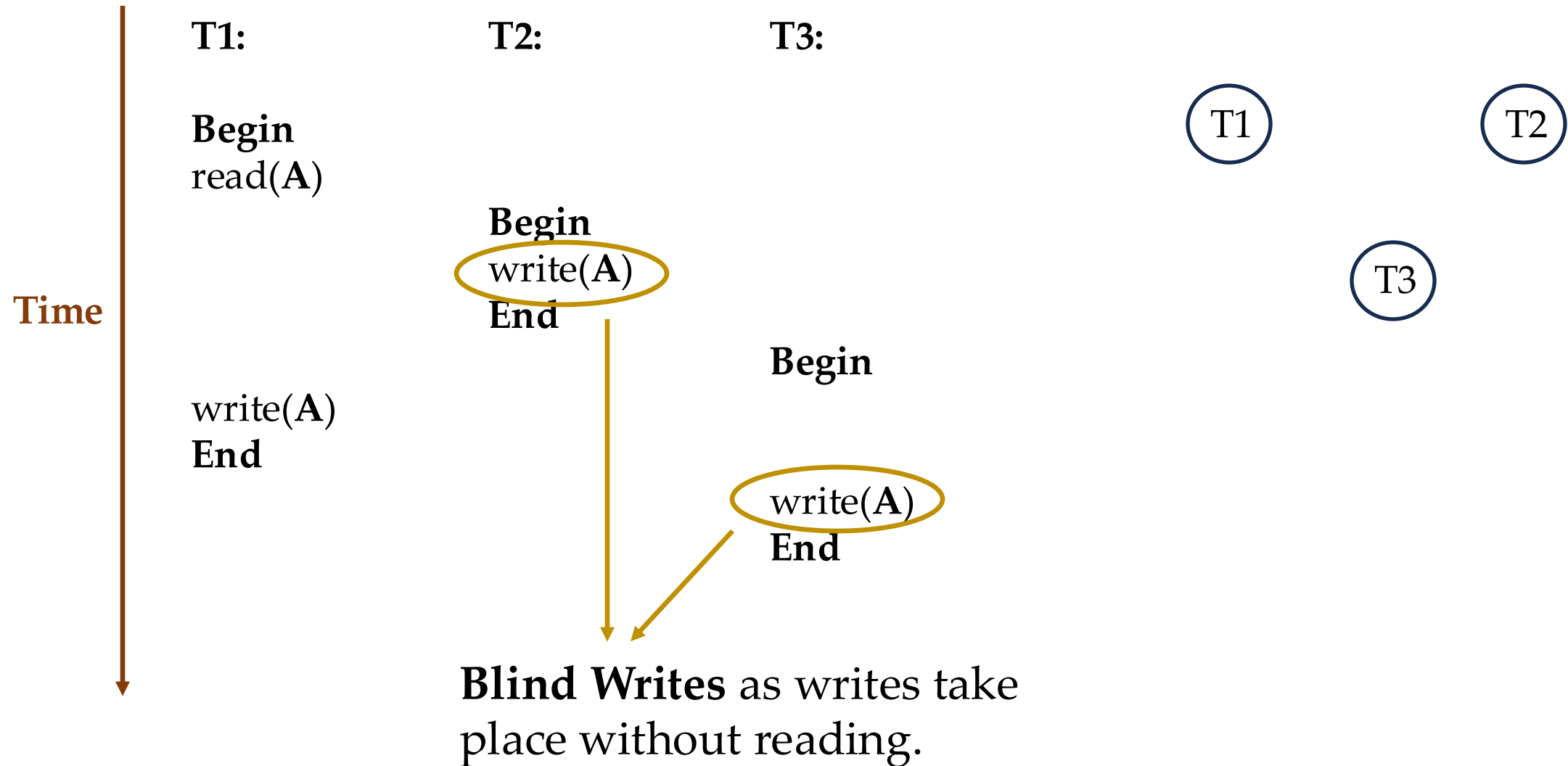
# View Serializability

- **View Serializability** essentially states that if the output of a schedule matches the expectation, then the schedule is view serializable even if it is not conflict serializable.
- **Formal Definition:** Schedules **S1** and **S2** are **view equivalent** if
  - T1 reads initial value of A in S1, then T1 also reads initial value of A in S2.
  - T1 reads value of A written by T2 in S1, then T1 also reads value of A written by T2 in S2 .
  - T1 writes final value of A in S1, then T1 also writes final value of A in S2 .

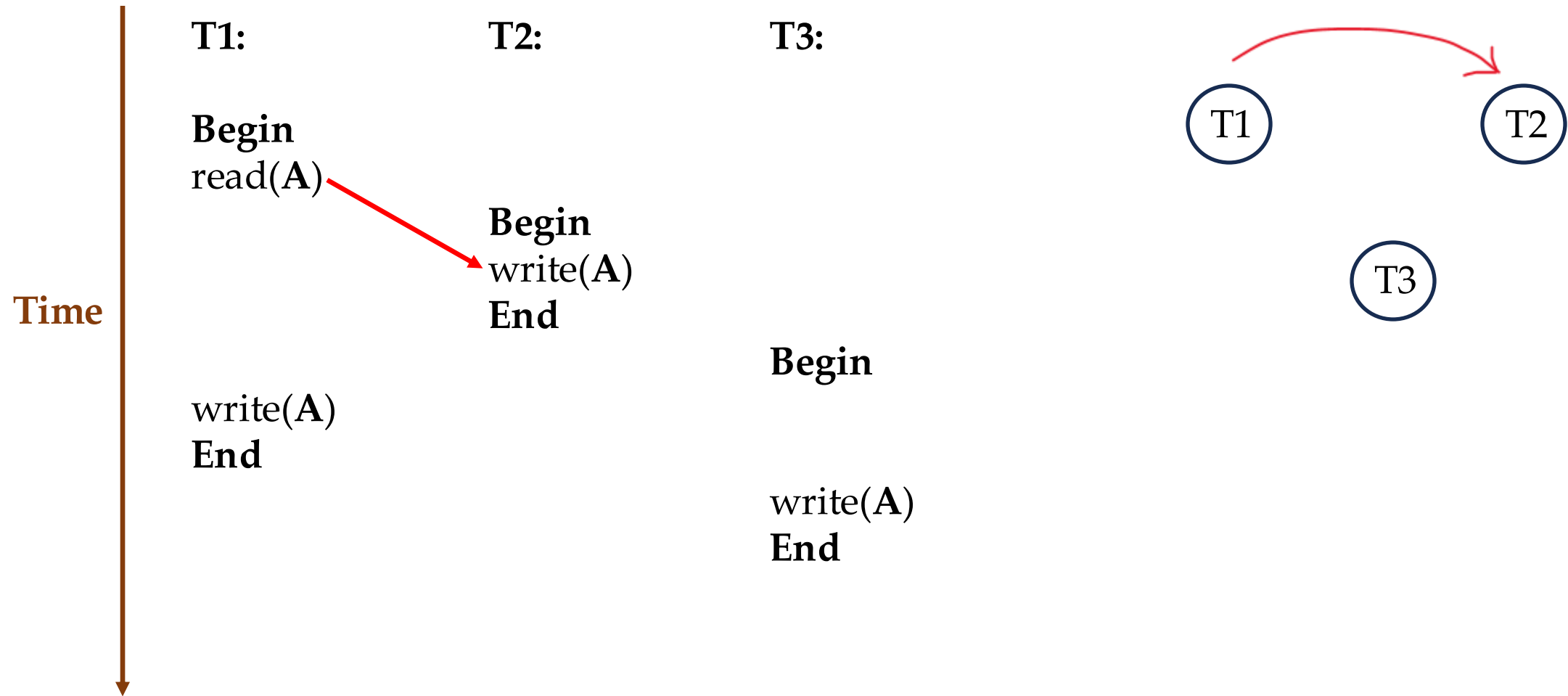
# View Serializability

- **View Serializability** essentially states that if the output of a schedule matches the expectation, then the schedule is view serializable even if it is not conflict serializable.
- **Formal Definition:** Schedules **S1** and **S2** are **view equivalent** if
  - T1 reads initial value of A in S1, then T1 also reads initial value of A in S2.
  - T1 reads value of A written by T2 in S1, then T1 also reads value of A written by T2 in S2 .
  - T1 writes final value of A in S1, then T1 also writes final value of A in S2 .
- View Serializability supports more schedules than Conflict Serializability.

# View Serializability

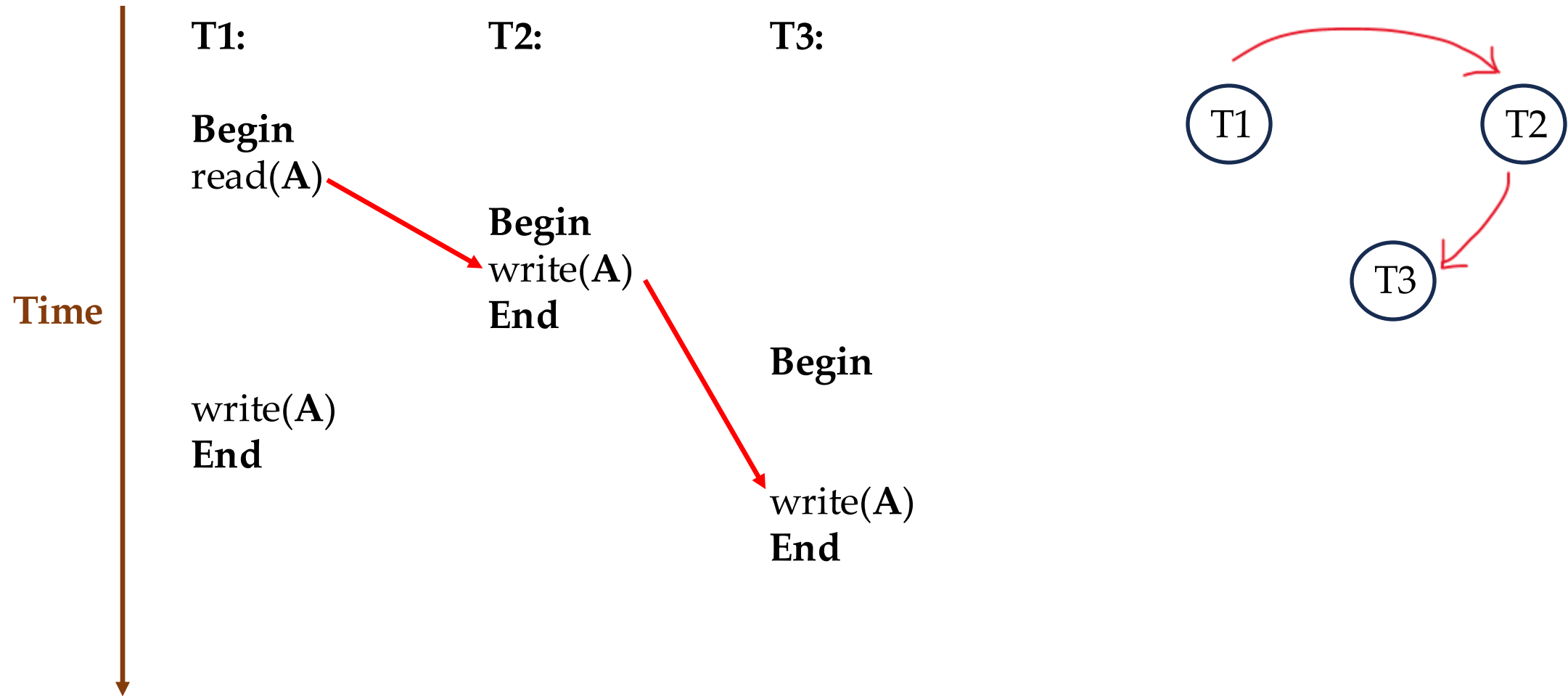


# View Serializability

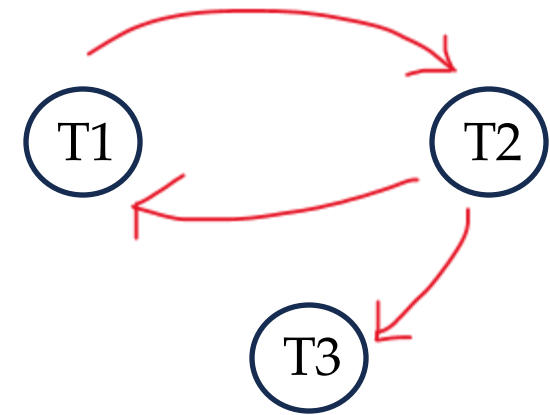
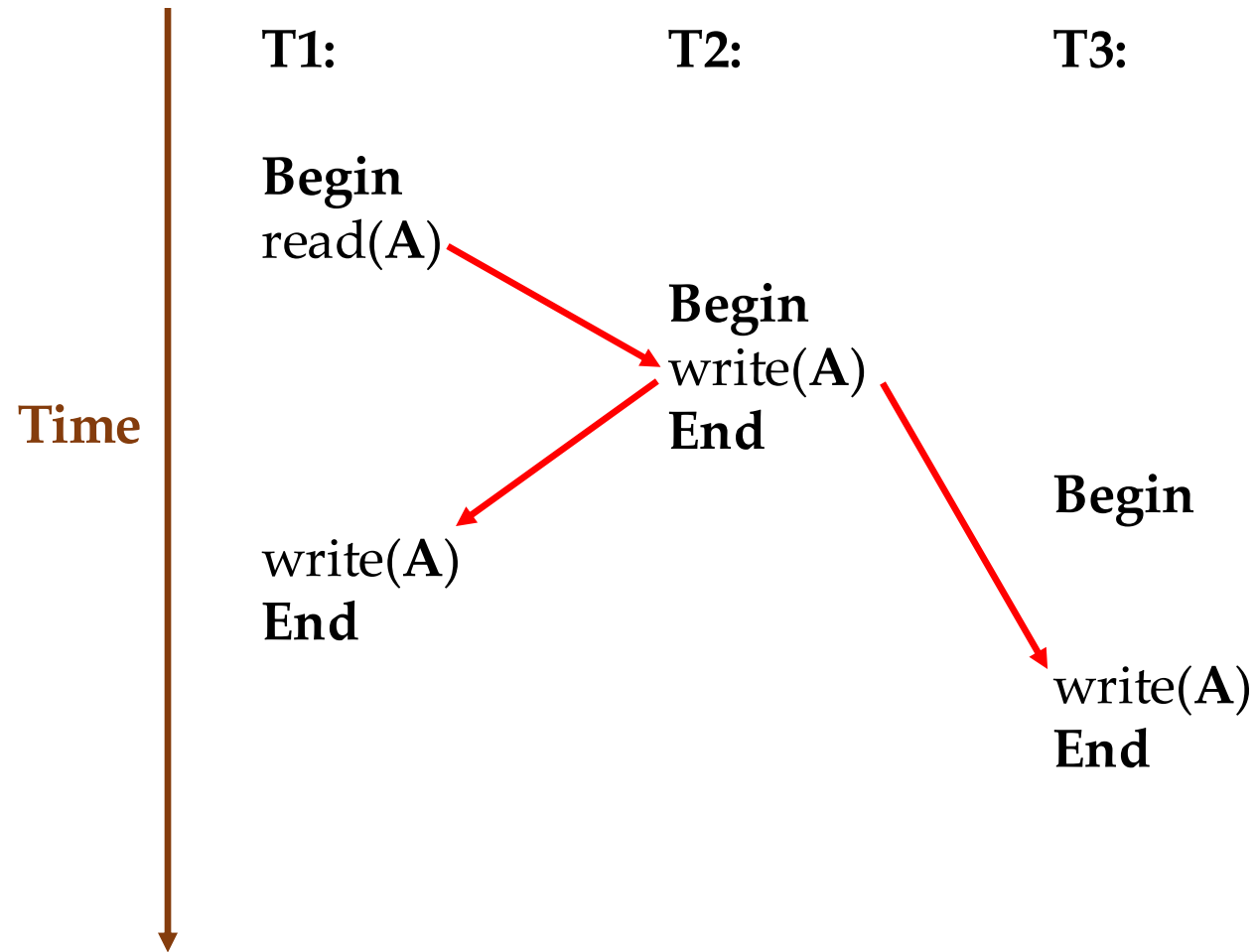




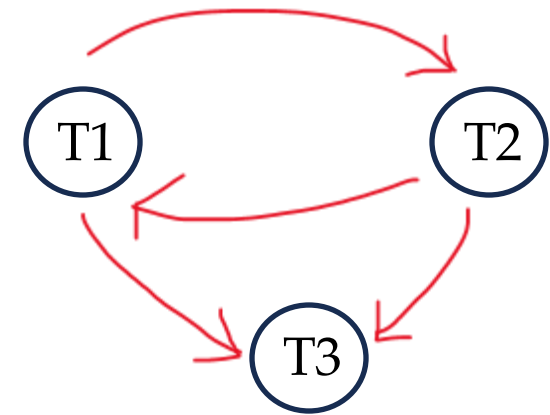
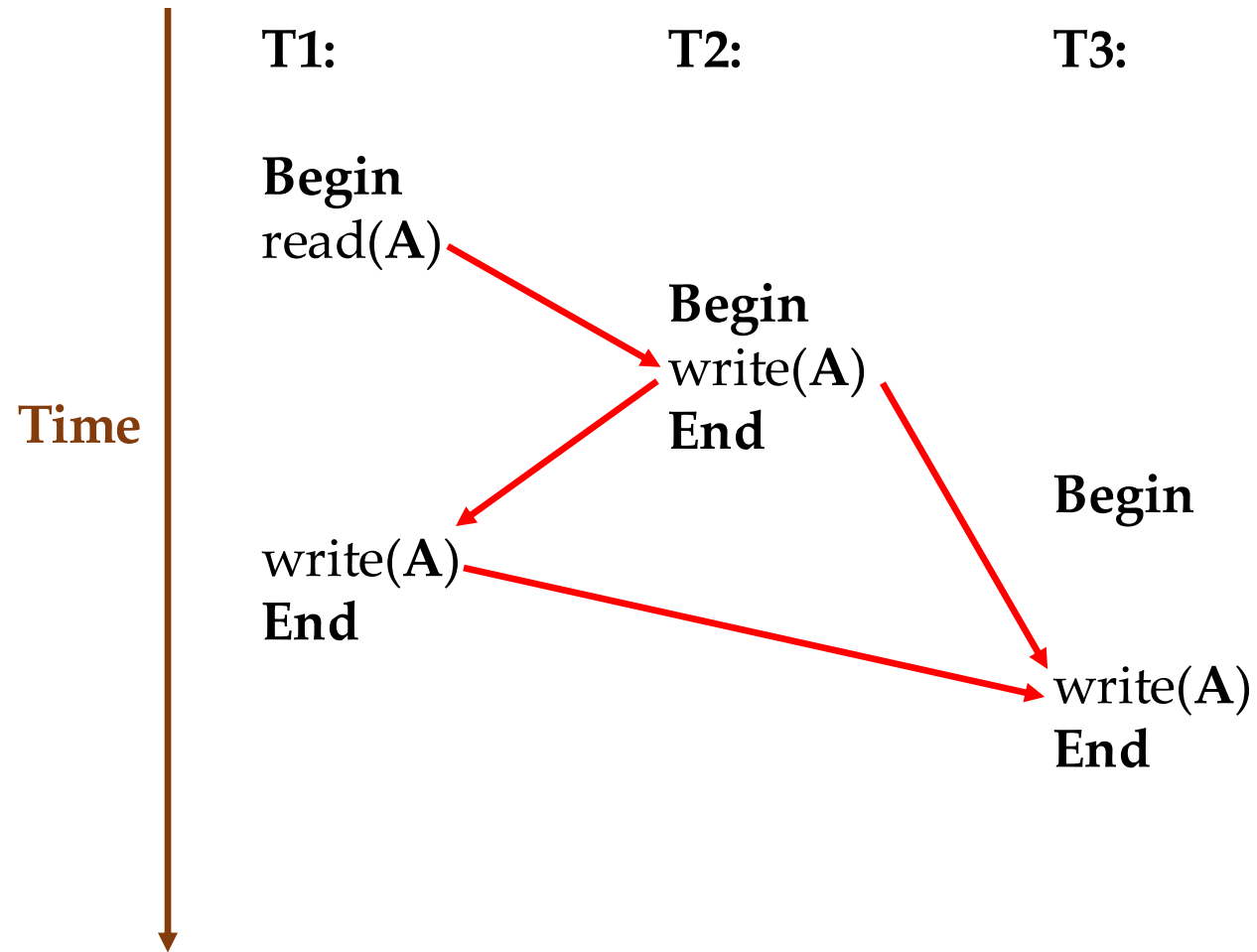
# View Serializability



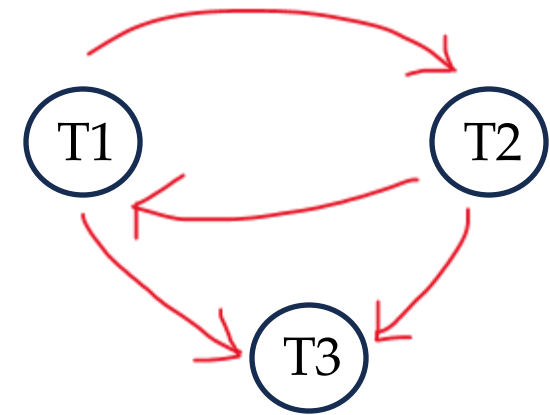
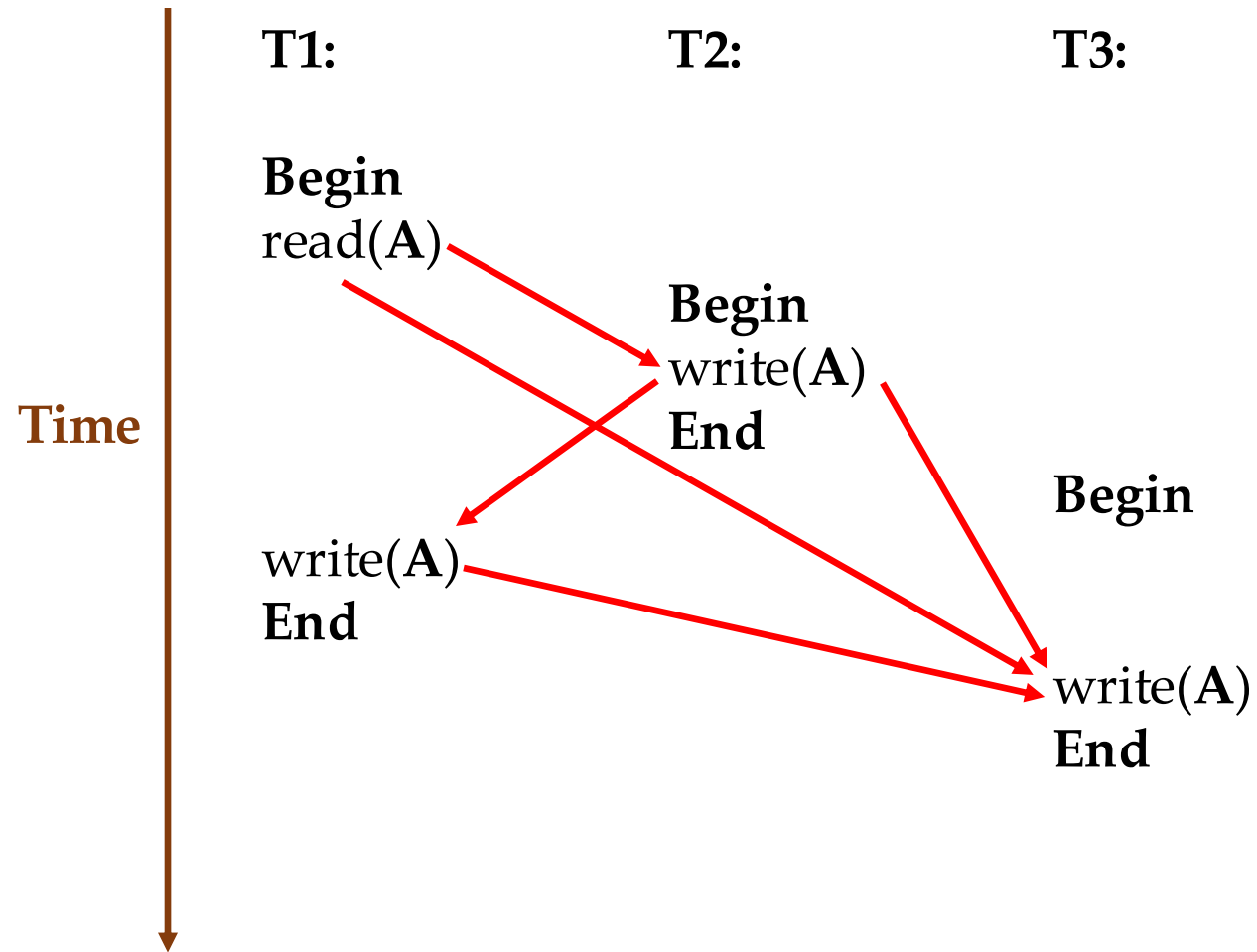
# View Serializability



# View Serializability



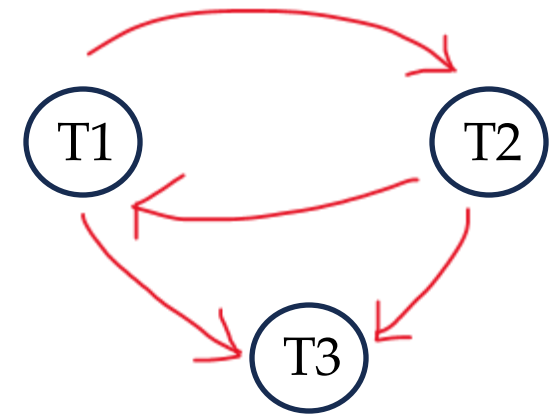
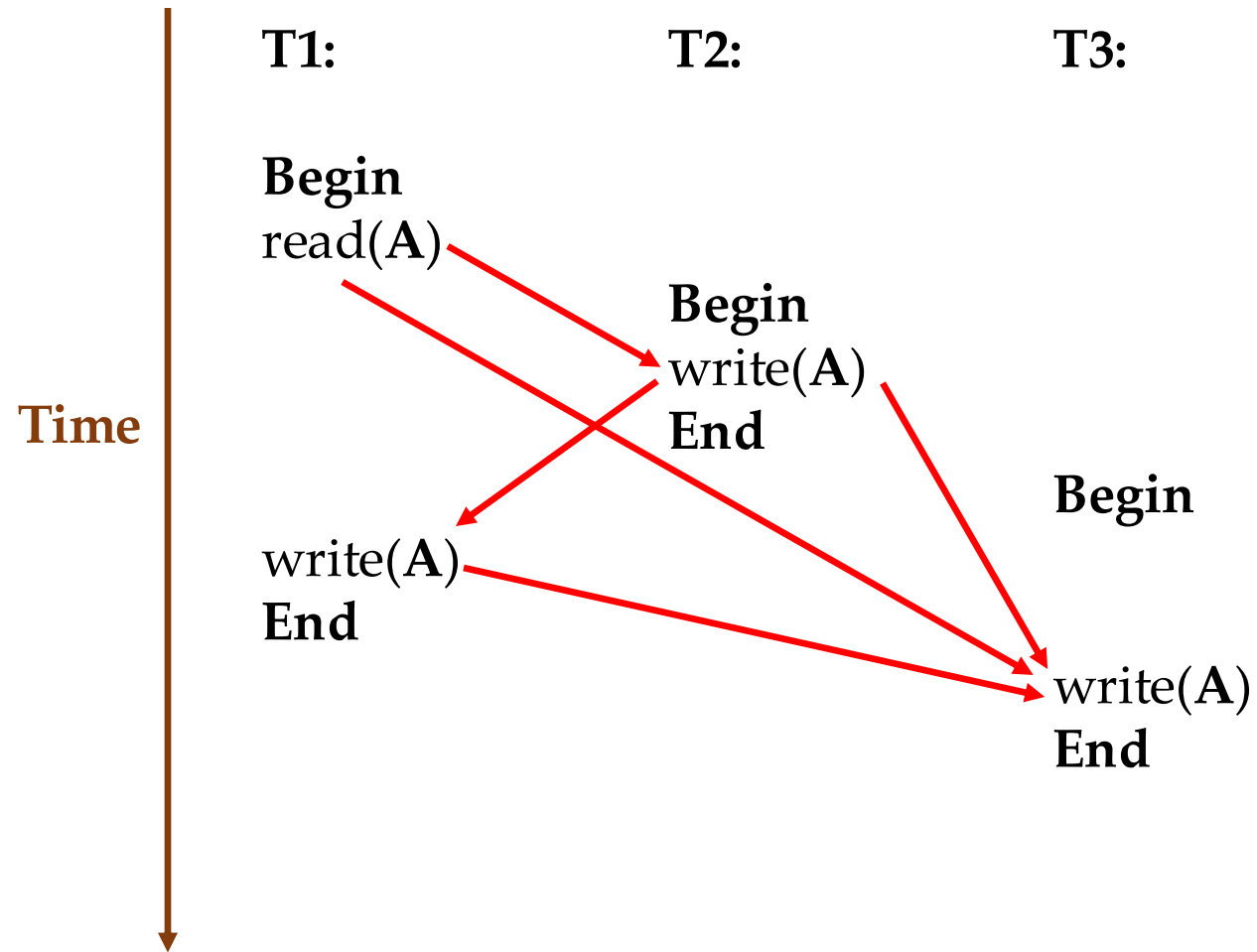
# View Serializability



Is this conflict serializable?

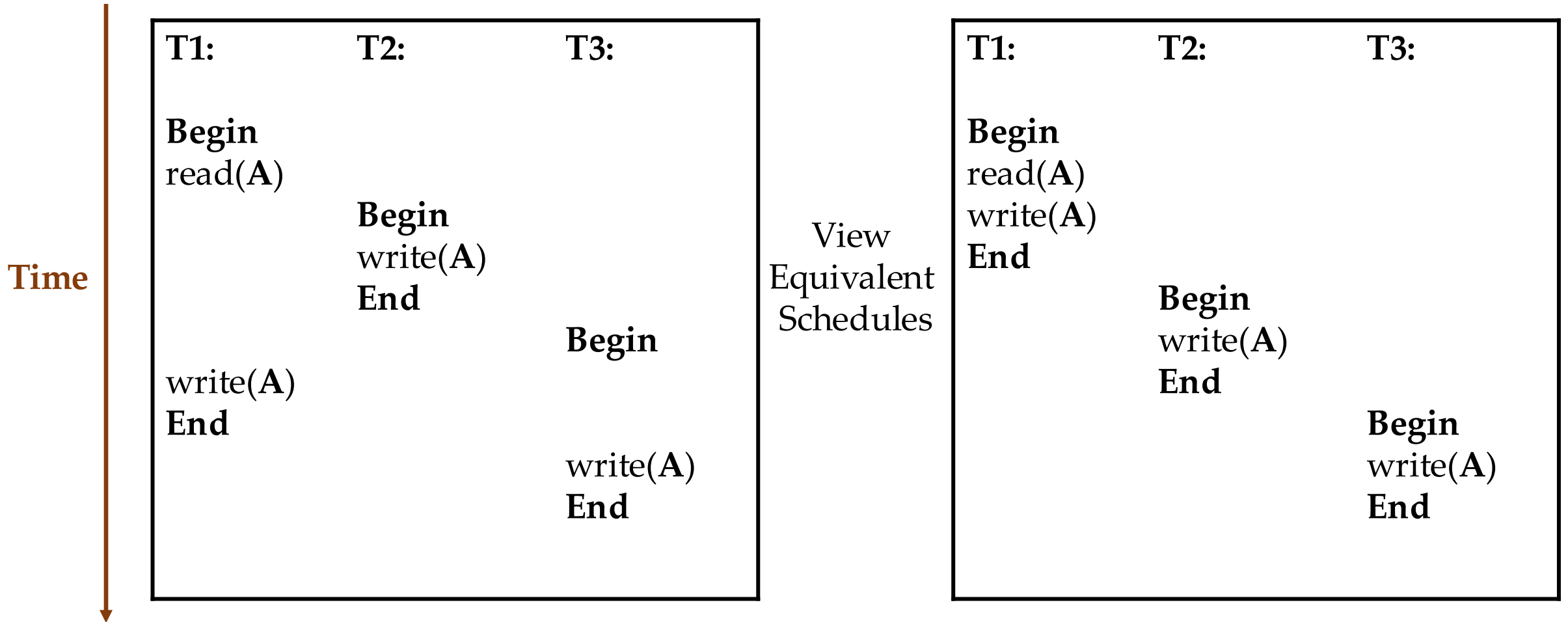
No!

# View Serializability



Is this view serializable?

# View Serializability



Is this view serializable? **Yes!**